Exploring Data through Ranked Entities

PhD thesis approved by
the Department of Computer science
Technische Universität Kaiserslautern
for the award of the doctoral degree

Doctor of Engineering (Dr.-Ing.)

to

Kiril Panev

Date of defense:
April 10, 2019

Dean:
Prof. Dr.-Ing. Stefan Deßloch

Reviewers:
Prof. Dr.-Ing. Sebastian Michel (TU Kaiserslautern)
Prof. Dr. Carsten Binnig (TU Darmstadt)

PhD committee chair:
Prof. Dr.-Ing Jens Schmitt

D 386

Abstract

Ranking lists are an essential methodology to succinctly summarize outstanding items, computed over database tables or crowdsourced in dedicated websites. In this thesis, we propose the usage of automatically generated, entity-centric rankings to discover insights in data. We present PALEO, a framework for data exploration through reverse engineering top-k database queries, that is, given a database and a sample top-k input list, our approach, aims at determining an SQL query that returns results similar to the provided input when executed over the database. The core problem consist of finding selection predicates that return the given items, determining the correct ranking criteria, and evaluating the most promising candidate queries first. PALEO operates on subset of the base data, uses data samples, histograms, descriptive statistics, and further proposes models that assess the suitability of candidate queries which facilitate limitation of false positives. Furthermore, this thesis presents COMPETE, a novel approach that models and computes dominance over user-provided input entities, given a database of top-k rankings. The resulting entities are found superior or inferior with tunable degree of dominance over the input set—a very intuitive, yet insightful way to explore pros and cons of entities of interest. Several notions of dominance are defined which differ in computational complexity and strictness of the dominance concept—yet, interdependent through containment relations. COMPETE is able to pick the most promising approach to satisfy a user request at minimal runtime latency, using a probabilistic model that is estimating the result sizes. The individual flavors of dominance are cast into a stack of algorithms over inverted indices and auxiliary structures, enabling pruning techniques to avoid significant data access over large datasets of rankings.

Zusammenfassung

Ranglisten sind eine essentielle Methodik um besondere, aus Datenbanktabellen berechnete oder auf speziellen Webseiten durch Crowdsourcing gesammelte, Datensätze, kurz und bündig zusammenzufassen. Wir schlagen in dieser Arbeit vor, automatisch generierte datensatz-zentrische Ranglisten zu verwenden, um Erkenntnisse aus Datensammlungen zu ziehen. Wir stellen PALEO, ein Framework zur Datenexploration durch Reverse Engineering von Top-K-Datenbankanfragen, vor. Für eine gegebene Datenbasis sowie eine beispielhafte Top-K-Liste als Eingabe zielt unser Ansatz darauf ab, eine SQL Anfrage zu bestimmen, welche der Eingabeliste ähnliche Ergebnisse erzeugt, wenn sie auf der Datenbasis ausgeführt wird. Im Kern besteht das Problem daraus, Selektionsprädikate zu finden, die durch die vorgegebenen Datensätze erfüllt werden, somit die korrekten Kriterien für die Ranglisten zu bestimmen, und die vielversprechendsten Kandidatenanfagen zuerst auszuwerten. PALEO operiert auf einer Teilmenge der Basisdaten und benutzt Stichproben, Histogramme, deskriptive Statistiken und führt außerdem Modelle ein, die die Passgenauigkeit von Kandidaten bewerten, um Fehler erster Art zu begrenzen. Darüber hinaus präsentiert diese Arbeit COMPETE, einen neuen Ansatz zur Modellierung und Berechnung von Dominanz von durch Benutzer eingegebenen Datensätzen für eine vorliegende Basis an Top-K-Ranglisten. Die zurückgegebenen Datensätze sind über- oder untergeordnet mit einem einstellbaren Grad an Dominanz über der Eingabemenge; eine sehr intuitive und doch aufschlussreiche Art und Weise zur Erforschung von Pro und Contra interessanter Datensätze. Verschiedene Dominanzbegriffe, welche sich in Berechnungskomplexität sowie Schärfe des Dominanzkonzepts unterscheiden und zum Teil ineinander enthalten sind, werden eingeführt. Es ist COMPETE möglich, denjenigen Ansatz zu wählen, der die Anfrage höchstwahrscheinlich mit kleinstmöglicher Verzögerung beantwortet, wofür COMPETE ein probabilistisches Modell zur Abschätzung der Ergebnisgröße einsetzt. Die einzelnen Abstufungen von Dominanz gehen in einer Vielzahl von Algorithmen über invertierten Indexen und Hilfsstrukturen auf und ermöglichen somit den Einsatz von Techniken zur signifikanten Verringerung des Datenzugriffs bei großen Mengen von Ranglisten als Datenbasis.

Contents

Chapter 1

Introduction

In the information age, the world continuously captures vast amounts of data, containing astronomical amounts of information. The data deluge is omnipresent and exponentially increasing; by 2020, it is estimated that for every person in the world, 1.7MB of data will be created *every second* [GR12]. Most of this complex and heterogeneous data is unstructured or semi-structured, making it difficult to extract information, with only a tiny fraction, less than 1%, being explored for analytical value. Ranked lists are a very intuitive and widely applicable concept to succinctly summarize outstanding items, capturing the relative performance of entities in a dataset. Essentially, rankings allow focusing on a small subset of an exhaustively full list, presenting the essence of the available data, worthwhile to look into. The concept of ranking is ubiquitous, it is used in nearly all domains; companies are ranked by revenue, mountains by height, football players by goals scored, etc. The clear need for creating novel methods for unearthing knowledge from data has stimulated various research in developing innovative ways for data exploration. Numerous dynamic data-driven applications focusing on helping users to intuitively discover information have been presented by the research community in the past years. In this thesis we propose efficient and effective methods that use rankings to explore entity-centric data. We present intuitive approaches for non-expert users, unfamiliar with the underlying data, to discover essential characteristics of not only their entities of interest, but also their peers in the dataset.

Exploratory queries, providing an immediate glance of some of the data properties, are becoming a prominent approach to understanding data. Recent approaches resort to examples as a query method for exploratory data analysis [MLVP17]. These methods exploit the intrinsic properties of the data to infer the information intend that the user has in mind, but maybe is unable to easily formulate. Particularly, the use of systems that reverse engineer SQL queries by using as input simple example result tuples were presented [TCP09, SPGW10, ZEPS13, SCC+14, PDCC15]. The discovered queries can reveal interesting properties of the input described by the different query constraints.

Additionally, various data exploration techniques rely on multiple queries, with the query results initiating the formulation of the next one [DP13, DPD14]. This data exploration paradigm is an essential component for a large number of discovery-oriented applications, e.g., financial analysis, in the medical domain, and genomics [IPC15].

Consider a user Alice who needs to make up her mind which smartphone to buy next. Alice is favoring iPhone XS, Samsung Galaxy S9, and Huawei Mate 20, in this order. She is interested in finding explanatory queries and in fact populated rankings that resemble this ranking. Assume she gets a ranking of {iPhone XS, Google Pixel 3, Samsung Galaxy S9, Huawei Mate 20} with constraints 'storage=64GB' and 'year=2018', ranked by 'battery lifetime'. What can she learn from that and how can she proceed? She can remove the constraint on the year to get additional offers, she also learned that the Google Pixel 3 appears feasible, too. Further, she changes the ranking criteria as 'battery lifetime' is not the most decisive criterion for her anyways, can distort the ranking slightly to see how generating queries are going to differ, etc. Thus, given the structure of the queries, Alice learns about the categorical constraints and ranking criteria used, i.e., what are the characteristics that make the smartphones top-performers. Moreover, given the computed rankings, Alice can further learn about other smartphones that perform perhaps even better, depending on how much it is allowed to deviate from the original input ranking.

Further, consider a user Bob who wants to buy a car. Bob is not a car enthusiast and does not know too much about cars yet, but there are a couple of models that he likes, say a BMW 118i and an Audi A3. Now, he would like to learn about other cars that outperform his favorites in some or all rankings where they appear together, i.e., the car models that are dominating his input. For instance, Bob can learn that Mercedes A220 is a model that outperforms the two input cars across all rankings. Thus, he learns that this model is always better than his favorites and that he should probably consider it as a potential alternative for buying. Moreover, he can examine what are the constraints and ranking criteria where the Mercedes is better, for example, in cars of type = Hatchback and ranked by power. He also wants to see which car models are dominated by his favorites, i.e., are always ranked worse, therefore he can disregard them as potential models of interest.

These two examples underpin the generic use of rankings for satisfying a user's information need in an intuitive and insightful exploratory fashion.

1.1 Problem Statement

In this thesis we propose two techniques that harness entity rankings for data exploration. First, we present an approach for reverse engineering top-k database queries, and second, we model and compute entity dominance in a set of top-k rankings. The two problems are treated independently and are defined more formally in the following.

Cars

Model	Year	TypeID	EngineID
Audi A3	2012	1	3
Audi A3	2014	1	5
...	...	...	...
BMW 118i	2014	1	2
BMW 118i	2010	2	14
...	...	...	...
BMW 325d	2013	50	30
Golf VII	2010	2	25
Mercedes A220	2014	1	1

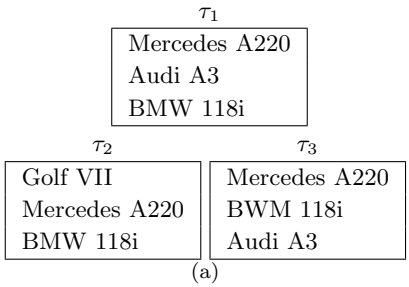
TypeData

TypeID	Type	Doors
1	Hatchback	5
2	Hatchback	3
...	...	...
50	Limousine	5

EngineData

EngineID	Power	Fuel
1	170	gas
2	144	gas
...	...	...
30	177	diesel

Figure 1.1: Sample database containing cars data



(a)

SELECT $L.e$, agg(value)
FROM R_1, R_2, ...
WHERE J_1 and J_2 and ...
and P_1 and P_2 and ...
GROUP BY $L.e$
ORDER BY agg(value) DESC
LIMIT k

(b)

Figure 1.2: Sample top-k rankings and query template

Consider a database D containing relations $R_1, R_2, \ldots, R_t$ with schema $R_i = \{A_{i_1}, A_{i_2}, \ldots\}$ and an input relation L that represents a ranked list of items. The task is to efficiently and effectively reverse engineer queries Q that, when executed over the database, compute result rankings that resemble the input list L. We focus on identifying top-k select-project-join queries of the form shown in Figure 1.2(b). The task can be broken down into identifying the different parts of the query: join predicates J_i, selection predicates P_i, and the ranking criteria agg($value$). Further, we introduce methods that identify not only the queries that exactly match the input list, but also find queries that produce rankings similar to the input ranking. A user-defined threshold θ is used to control how similar the found queries, respectively their results, should be, relative to the input L. For comparing the top-k lists, we use Spearman's Footrule [FKS03], a prominent distance measure used for comparing top-k lists.

For instance, consider the database shown in Figure 1.1, containing data about cars. The database contains information about cars, such as model, year produced, body type, and engine type. Now consider the ranking τ_1 shown in Figure 1.2(a), containing an ordered list of car models, without any additional info. Considering the database in Figure 1.1, we can determine that the top-k list can be generated with a query containing a natural join over the three tables, having year=2014 and type=Hatchback as selection predicates, with max(power)

as ranking criteria.

Having a small input list makes it difficult to extract meaningful properties in order to identify the various parts of the query. Furthermore, the presence of false positive and false negative candidate queries can significantly impair system performance due to possibly many (potentially expensive) candidate query evaluations. We propose pruning strategies and smart candidate query verification to deal with these issues. Moreover, identifying similar queries substantially increases the search space, thus, we embed distance-measure specific bounds into the query-generation process.

Consider further a dataset T of rankings τ_i. Each ranking has a domain D_{τ_i} of items it contains and there are no duplicate items nor ties in the rankings. Given a set of input entities I, our objective is to efficiently and effectively determine all entities e that dominate the entities in I, i.e., are ranked higher in all lists where they co-occur. We also want to identify the set of dominated entities, that are ranked lower than $e_i \in I$ in all lists τ they appear together. We further extend the problem by relaxing the criterion of an entity dominating all entities in I to at least some of the input entities, as well as dominating across all common rankings to a certain portion of the rankings.

For example, consider the rankings shown in Figure 1.2(a) and the input set $I = \{$Audi A3, BMW 118i$\}$. We can determine that Mercedes A220 is a model that dominates the input I, since it is ranked higher than both Audi A3 and BMW 118i in all the rankings.

The problem is challenging given the multi-entity input and the subtle nuances that the dominance relationship entails. We distinctly refine the definition of dominance and propose algorithms that rely on suitable index structures and pruning techniques to avoid significant data access over large sets of rankings.

1.2 Contributions

With this work, we make the following contributions to the area of data exploration:

- We present PALEO—a novel approach to explore data through reverse engineering top-k OLAP queries. We propose an approach that mainly operates on a subset of the data, held in memory, and further uses data samples, histograms, and simple descriptive statistics to identify potentially valid queries, that generate the input list.

- We extend the framework to reverse engineer approximate matches within a distance measure threshold and present techniques using distance-measure induced bounds for early pruning of candidates.

- We present a probabilistic reasoning that prioritizes the execution of candidate queries by the likelihood that they compute the input ranking.

Together with an iterative approach of skipping unpromising queries dynamically at query validation time, this results in efficient discovery of valid queries, confirmed by an extensive experimental evaluation.

- We complement PALEO by proposing an efficient approach for identifying join constraints, by making use of database meta-data and appropriate data structures.
- We present COMPETE—an efficient, effective, and extensible approach for exploring data through entity-centric rankings that computes dominating and dominated entities. We define the entity dominance relationship over a dataset of rankings and consider different types of dominance given a set of input entities which differ in the strictness of the dominance concept.
- We propose a model for estimating the number of dominating entities for a set of input entities, used for looking the most promising type of dominance to satisfy a user request at minimal runtime latency. Further, we propose a stack of algorithms utilizing inverted indices and auxiliary structures that harness different performance optimizations.

1.3 Publications

The content of this thesis has been published in various peer-reviewed papers in conferences and workshops. Following, we give a high level overview of the publications.

In [PM16], we have presented the initial PALEO approach, working on a single relation, described in Chapter 4. PALEO shows a compartmentalized approach in reverse engineering the different components of a top-k query, with focus on avoiding false positives and early elimination of unpromising candidates.

- Kiril Panev and Sebastian Michel. Reverse Engineering Top-k Database Queries with PALEO. 19th International Conference on Extending Database Technology (EDBT), 2016.

In [PMM16], we have addressed the problem of discovering queries that, given a user-defined similarity threshold, produce lists similar to the input ranking. We show how to incorporate distance bounds into the methods in order to keep the search space tractable.

- Kiril Panev, Evica Milchevski, and Sebastian Michel. Computing Similar Entity Rankings via Reverse Engineering of Top-k Database Queries. Workshop on Keyword Search and Data Exploration on Structured Data (KEYS), 2016.

A full-fledged prototype implementation of PALEO has been demonstrated at VLDB 2016 [PMMP16].

- Kiril Panev, Sebastian Michel, Evica Milchevski, and Koninika Pal. Exploring Databases via Reverse Engineering Ranking Queries with PALEO. 42nd International Conference on Very Large Databases (VLDB), 2016.

In [PWM17], we extend the framework with an approach to identify join constraints that efficiently classifies and prioritizes promising candidate join queries. This approach complements the initial PALEO framework into supporting top-k select-project-join queries and is described in Chapter 5.

- Kiril Panev, Nico Weisenauer, and Sebastian Michel. Reverse Engineering Top-k Join Queries. 17. GI-Fachtagung für Datanbanksysteme in Business, Technologie und Web (BTW), 2017.

In [PM18], we have presented COMPETE—a framework that models and computes dominance over user-provided input entities, given a database of top-k rankings. In this work, described in Chapter 6, we have defined several notions of dominance, which differ in computational complexity and strictness of the dominance concept, and propose a stack of algorithms employing pruning techniques over inverted indices and auxiliary data structures.

- Kiril Panev and Sebastian Michel. Exploring Pros and Cons of Ranked Entities with COMPETE. Workshop on Exploratory Search in Databases and the Web (ExploreDB), 2018.

1.4 Outline of the Thesis

The thesis is organized as follows. Chapter 2 presents an overview of basic concepts, methods, and structures used throughout our work. Chapter 3 discusses work related to the data exploration techniques described in the thesis. Chapter 4 introduces the PALEO framework for reverse engineering top-k database queries. It describes methods that identify the different parts of a query and the impact of allowing discovery of queries that produce results similar to the input ranking. Chapter 5 presents a complementary approach for reverse engineering top-k join queries. Chapter 6 presents COMPETE, our framework for exploring entity-centric data, where we model and compute dominance of entities in a dataset of top-k rankings. Chapter 7 presents the conclusion from this thesis and discusses future work.

Chapter 2

Background and Preliminaries

In this chapter, we present background knowledge necessary for understanding the work described in this thesis. It discusses some of the main concepts, algorithms, and evaluation measures in the area of information retrieval. Then, it introduces top-k rankings, data structures for their indexing, and distance functions frequently used in similarity search. Further, fundamental technologies for data analysis like online analytical processing and knowledge discovery techniques are described. Finally, the concept of skyline query processing is presented as a core principle in finding interesting data points in a database.

2.1 Information Retrieval

The field of *information retrieval (IR)* studies the problem of satisfying user's information need within a large collection of data resources [MRS08]. The data processed using IR techniques is unstructured or semistructured, usually collected from the Internet, e.g., web documents containing text with limited explicit markup, twitter data, crowdsourced rankings, etc. The lack of structure in the documents makes them difficult to understand by computers, thus IR techniques usually return multiple results, which often are ranked by a scoring function how well they match the information need. In modern times, the information retrieval task is synonymous with the word *search*, with *search engines* being the most common application. In searching a collection of documents, the user information need is specified in terms of a *query* containing a few search terms and the result is a ranked set of documents.

d_1 : the history book.
d_2 : a book of history.
d_3 : history of writing a book.

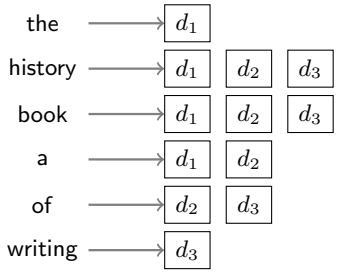


Figure 2.1: Example of an inverted index

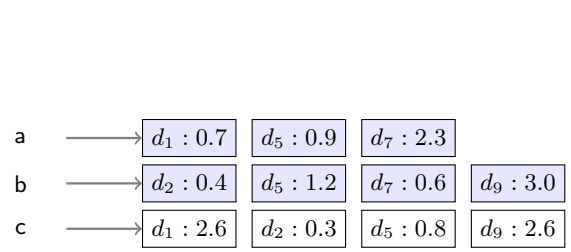
2.1.1 Inverted Index

The *inverted index* is one of the most prominent data structures used in searching a collection of documents [ZM06]. It maps terms to a posting list containing information about the occurrences of the corresponding term in the document collection. The inverted index reduces the search space by eliminating the documents that do not contain a query term. Moreover, it allows for efficiently identifying those documents that have one or more terms in common with the query terms. Depending on the application task, the posting lists in the inverted index can contain additional information such as the position of the term in the document or a score, e.g., the number of times the term occurs in the document, or weight reflecting the importance of the term in the document. Additionally, depending on the retrieval model, the posting list can be sorted according to any of the information it contains.

A sample document collection and its corresponding inverted index is shown in Figure 2.1. Looking up in the inverted index, for instance, can reveal that writing appears only in the document d_3.

2.1.2 Processing Keyword Queries

Query processing details can vary in different search engines and the essential ideas depend on the type of retrieval, with boolean and ranked retrieval representing the most prominent models [MRS08]. Querying in the boolean retrieval model relies on using a precise language and operators, while in ranked retrieval users commonly use free text queries and the system ranks the search results according to query relevance. Two of the most common approaches to ranked retrieval are *term-at-a-time* and *document-at-a-time* query processing [BCC10].



Accumulators

document	score
d_1	0.7
d_2	0.4
d_5	2.1
d_7	2.9
d_9	3.0

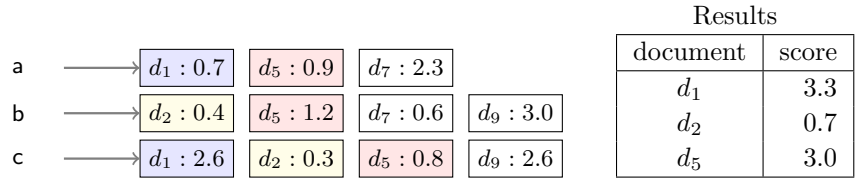
Figure 2.2: Example with TaaT query processing

Term-at-a-Time

In term-at-a-time (TaaT) query processing, the query terms are processed successively one at any given time. After complete processing of a term's posting list, the evaluation continues with one of the unprocessed terms in the query. This approach maintains a set of document score accumulators and for each inspected posting, the score value for the respective accumulator is updated according to the score in the posting list. Once all documents are processed, the accumulators contain the final scores of the documents regarding the query.

Figure 2.2 shows an example with TaaT query processing in evaluating a query $q = [\mathsf{a}, \mathsf{b}, \mathsf{c}]$. The figure depicts the state of the document accumulators after evaluating the posting lists of the terms a and b. The accumulators reflect the scores at this point of the processing, computed by accumulating the values in the corresponding postings. In the next step of the evaluation, the last posting list (of the term c) is processed and the accumulator scores updated to the final scores of the documents. For instance, the final score of d_1 is 3.3.

A common optimization for conjunctive queries is to process the query terms in ascending order of their document frequency (i.e., the length of their posting list), thus keeping the number of accumulators and memory low. Note that if q is processed as a conjunctive query, there is no need to create accumulators for all documents. A conjunctive query returns as results only the documents that contain all query terms. After processing the rarest term (i.e., a), accumulators for the documents in the posting list of a are created and these documents are considered as a candidate set. In processing the next query term, only the postings that contain documents in the candidate set are considered and their scores are updated. If for a certain candidate there is no matching document in the posting list, then this candidate document can be removed from the list of potential results. For instance, the document d_1 can be removed as a potential result, since it is not present in the posting list of b. Similarly, d_9 will not be considered as a candidate document at all.



Results

document	score
d_1	3.3
d_2	0.7
d_5	3.0

Figure 2.3: Example with DaaT query processing

Document-at-a-Time

In document-at-a-time (DaaT) query processing, the query terms are processed in parallel, by intersecting their corresponding posting lists. This approach completely evaluates and scores a single document by examining the occurrence of the query terms in it. All posting lists are iterated at the same time and aligned on a matching document. The advancement is done on the posting list with the lowest current document identifier, hence the posting lists need to be document-ordered.

Figure 2.3 depicts an example with DaaT query processing. All posting lists are processed concurrently and the figure shows the state after computing the score for document d_5, with documents d_1, d_2, and d_5 being completely evaluated for the entire query. In the next step, the posting lists will be aligned on d_7 and its score computed. The query evaluation finishes when all posting lists have been processed entirely.

In processing conjunctive queries, the processing of the posting lists can be optimized by using skip pointers [BCC10]. The skip pointers reference document identifiers further along the list and are used as shortcuts to avoid processing postings with documents that will not be in the results. Using these pointers it can be examined whether or not a portion of the inverted list can be skipped by checking if the document identifier of the pointer is less than the identifier of the document that should be evaluated.

2.1.3 Results Quality

An information retrieval system needs to be efficient in satisfying the user's information need. The efficiency is measured with the query response time and the aim is to keep the response time very low. However, in addition to fast query processing, producing results of high quality, relevant to the information need, is essential. Thus, IR-systems need to be not only efficient, but also effective. Measuring the effectiveness of a system is done by establishing a ground truth for a query and performing relevance assessment by classifying the documents in the dataset into relevant and non-relevant to the given query. The two most common and basic measures for information retrieval effectiveness are *precision* and *recall* [MRS08].

	relevant	non-relevant
retrieved	true positives (tp)	false positives (fp)
not retrieved	false negatives (fn)	true negatives (tn)

Table 2.1: Document relevance in an IR-system

Precision and Recall

When a system returns a set of documents as a query result, precision is computed as the fraction of retrieved documents that are relevant.

$$\text{precision} = \frac{\#(\text{relevant documents retrieved})}{\#(\text{retrieved documents})}$$

Recall is computed as the fraction of relevant documents that are retrieved.

$$\text{recall} = \frac{\#(\text{relevant documents retrieved})}{\#(\text{relevant documents})}$$

The notions in Table 2.1 are commonly used in making the computation of precision and recall more clear:

$$\text{precision} = \frac{\text{tp}}{(\text{tp+fp})} \qquad \text{recall} = \frac{\text{tp}}{(\text{tp+fn})}$$

Note that here a false positive is the system retrieving a document that is non-relevant, i.e., the document should not have been retrieved, while a false negative is the system failing to retrieve a document that is relevant, but should have been retrieved. Moreover, the false positives have an effect on the precision, while the false negatives have an effect on the recall of a system.

Precision and recall are set-based measures and are used in retrieval of unordered sets of documents. However, the standard in modern IR-systems is producing results that are ranked according to how well they are related to a query. Thus, extending the set-based measures is necessary in order to evaluate the ranked results.

Precision@k and Recall@k

Precision@k considers the top-k portion of the ranked result documents retrieved by the system and is computed as the fraction of relevant documents within the top-k results:

$$\text{precision@k} = \frac{\#(\text{relevant documents in top-k})}{k}$$

Recall@k is computed as the fraction of relevant documents within the top-k results:

$$\text{recall@k} = \frac{\#(\text{relevant documents in top-k})}{\#(\text{relevant documents})}$$

For instance, precision@10=0.4 signifies that there are 4 relevant documents in the top-10 retrieved results, while recall@10=0.4 shows that 40% of all relevant documents are within the top-10 results. These measures are useful in applications where only the top-k results are going to be examined.

2.2 Top-k Queries

Ranked query processing is essential in many applications that involve massive amounts of data. *Top-k queries* retrieve only the k results with the best scores and are crucial in various domains such as web search, relational databases, and distributed systems [IBS08]. The data objects relevant to the query are assigned a score using a scoring function—usually defined as an aggregation function over partial scores—with the top-k scored objects shown to the user as results. Thus, top-k query processing provides an easily comprehensible way to focus on the objects with outstanding performance.

2.2.1 Taxonomy of Top-k Queries

The top-k processing techniques can be categorized based on multiple design dimensions such as query model, data access, implementation level, and supported ranking functions [IBS08]. For this thesis, the most significant is the query model dimension, therefore it will be presented in more detail. The other dimensions are discussed briefly for completeness.

Query Model

The query model specifies which data objects need to be scored by the top-k processing techniques. We discuss three different query models: (i) top-k selection query, (ii) top-k join query, and (iii) top-k aggregate query.

Top-k Selection Query. In this query model, the base tuples contain the scores that will be used in the ranking. A top-k selection query will return as results the k-highest scored tuples. The final score of a tuple can also be a result of a user-defined function that aggregates multiple attributes from the same tuple.

More formally, given a relation R with schema $R = \{A_1, \ldots, A_n\}$ and scoring predicates, $s_1, \ldots, s_m$ defined on the attributes. Further, let $score(t) = score(s_1(t), \ldots, s_m(t))$ be the overall score of a tuple $t \in R$. A top-k selection query returns as results the o tuples from R with highest $score$ values ranked by the ranking criterion $score$.

Figure 2.4(a) shows the query template for the top-k selection query model.

Top-k Join Query. In this model, the scores that are used in the ranking are attached to join results instead of base tuples. A top-k join query joins

SELECT attributes
FROM R
WHERE predicate
ORDER BY $score(s_1, \ldots, s_m)$
LIMIT k

(a) Selection query

SELECT attributes
FROM $R_1, \ldots, R_n$
WHERE join_predicate
ORDER BY $score(s_1, \ldots, s_m)$
LIMIT k

(b) Join query

SELECT $g_1, \ldots, g_l$, agg()
FROM R
GROUP BY $g_1, \ldots, g_l$
ORDER BY agg()
LIMIT k

(c) Aggregation query

Figure 2.4: Top-k query templates

multiple relations on some join condition and computes the scores of the join results using a scoring function. The ranking criteria can involve attributes from multiple relations. The top-k join results ranked by the scoring function are returned as results.

Formally, given multiple relations $R_1, \ldots, R_n$, a top-k join query joins the relations and returns as result the top-k join results with highest combined scores. The scores of the join results are computed based on a scoring function $score(t) = score(s_1(t), \ldots, s_m(t))$, where $s_1, \ldots, s_m$ are scoring predicates over the join results.

The query template for the top-k join query model is shown in Figure 2.4(b).

Top-k Aggregate Query. In this model, the scores that are used in the ranking are computed for groups of tuples instead of individual tuples. With a top-k aggregate query, for each group of tuples, a score is computed using a group aggregate function (e.g., sum, avg) and the k groups with highest scores are returned as results.

Formally, given a relation R, a set of grouping attributes $g_1, \ldots, g_l$, and an aggregate function agg() used on each of the groups. A top-k aggregate query returns the k highest ranked groups according to agg().

Figure 2.4(c) shows the query template for the top-k join query model.

Note that the different query models can be combined into more complex queries, for instance, a top-k selection join query with aggregation.

Data Access

The top-k processing techniques can be classified according to the data access methods in the underlying data sources. The classification is done on the degree of allowed random accesses, as follows: (i) sorted and random accesses, (ii) no random accesses, and (iii) sorted access with controlled random probes.

Prominent example methods include the Threshold algorithm (TA) [FLN01] for sorted and random access, the No-Random-Access (NRA) method [FLN01] for no random access, and the Rank-Join algorithm [IAE04] for sorted access with controlled random probes.

Implementation Level

Depending on the integration level with the database engine, a top-k processing method can be implemented in (i) the application or (ii) the query engine level. The methods in the application level category can rely on specialized indices or materialized views, however, the top-k processing is done outside the database engine (e.g., [CBC+00, HKP01]). The methods in the query engine level can introduce new query operators and modify the query engine to obtain top-k processing and optimization (e.g., [IAE04, LCIS05, LCI06]).

Query and Data Uncertainty

Large amounts of data in some query processing applications might lead to considering sacrificing accuracy in favor of performance and report approximate answers. Depending on query and data uncertainty, top-k processing techniques can be classified as: (i) exact methods over certain data, where both queries and data are deterministic (e.g., [FLN01]), (ii) approximate methods over certain data, where the methods return approximate answers, but operate on deterministic data (e.g., [ARSZ03, TSW05]), and (iii) uncertain data, where top-k processing methods based on uncertainty models work on probabilistic data (e.g., [RDS07, SIC07]).

Ranking Function

The ranking function greatly influences the design of the top-k processing methods. Depending on the restrictions applied on the ranking function, the methods can be distinguished into methods with: (i) monotone ranking function, (ii) generic ranking function, and (iii) no ranking function. Most of the top-k methods assume monotone ranking functions which allow for efficient retrieval, with TA and NRA [FLN01] being the most prominent methods. The methods with generic ranking function consider arbitrary non-monotone functions and are optimized based on query predicates [ZHC+06]. Skyline queries [BKS01] return the data objects that are not dominated by any other objects in the database restricted to a set of dimensions and belong to the queries with no ranking function. We discuss skyline queries in more detail in Section 2.6.

2.2.2 Prominent Top-k Algorithms

The Threshold Algorithm (TA). Fagin et al. [FLN01] introduce one of the most prominent top-k processing methods in literature. The algorithm assumes

method: **thresholdAlgorithm**

```
1  scan posting lists L_i (round-robin)
2  consider d = cd_id(i) in posting list L_i
3  high(i) = cscore(i)
4  /* compute score(d) */
5  if d ∉ top-k then
6     look up score_j(d) in L_j for all j ≠ i
7     score(d) = agg{score_j(d) | j = 1...|q|}
8  /* update top-k */
9  if score(d) > min-k then
10    add d to top-k and remove min-score d'
11    min-k = min{score(d') | d' ∈ top-k}
12 /* update upper bound from current scan line */
13 ub = sum{high(i) | i = 1...|q|}
14 if ub ≤ min-k then return top-k
```

Algorithm 1: Threshold algorithm (TA)

that the cost of sorted and random accesses is the same and does not restrict the number of random accesses. It processes multiple score-ordered lists L_i which represent different rankings of the same set of data objects. An entry in L_i is of the form $(d : score_i(d))$, i.e., each list contains the object and its score in the list. The algorithm determines the top-k objects that have the highest aggregated score from all lists.

Algorithm 1 describes the details of TA. It scans the score-ordered posting lists L_i and looks up immediately the scores for each object d in the remaining lists (Lines 6–7 in Algorithm 1). Furthermore, it maintains an upper bound ub for the overall score of unseen objects, computed by aggregating the partial scores of the last seen objects in the different posting lists (Line 13 in Algorithm 1). The algorithm returns results once it finds k objects with aggregated score larger than the upper bound.

No-Random-Access Algorithm (NRA). Random accesses can be very expensive and some systems may not support random accesses at all. Therefore, Fagin et al. [FLN01] additionally introduce a method that only exploits sorted accesses. The NRA algorithm computes the top-k objects by using bounds computed over their exact scores.

Algorithm 2 shows the details of the NRA algorithm. Similarly to TA, it scans the score-ordered posting lists L_i, however it does not look up the exact scores in the remaining lists and for each object d, it rather computes lower (worst score) and upper (best score) bounds (Lines 6–7 in Algorithm 2). The worst score of an object d is computed by aggregating the object's known scores, while the best score of d is obtained by aggregating the object's known scores and the maximum possible values of its unknown scores, which are the

method: **noRandomAccess**

1 scan posting lists L_i (round-robin)
2 consider $d = cd_{id}(i)$ in posting list L_i
3 $E(d) = E(d) \cup i$ /* seen scores for d */
4 $high(i) = cscore(i)$
5 /* compute bounds for d */
6 worstscore(d)=agg$\{score_j(d) \mid j \in E(d)\}$
7 bestscore(d) = agg$\{$worstscore(d), agg$\{high_j \mid j \notin E(d)\}\}$
8 /* update top-k */
9 **if** worstscore(d) > min-k **then**
10 add d to top-k and remove min-score d'
11 min-k = $min\{$worstscore(d') $\mid d' \in$ top-k$\}$
12 **else if** bestscore(d) > min-k **then**
13 add d to candidates
14 /* update upper bound from current scan line */
15 ub = max$\{$bestscore(d') $\mid d' \in$ candidates$\}$
16 **if** $ub \leq$ min-k **then return** top-k

Algorithm 2: No-Random-Access algorithm (NRA)

last seen scores in the corresponding posting list. In this way, a top-k result can be returned even if its exact score is unknown. The top-k contains the k objects with largest worst score (Lines 9–10 in Algorithm 2). Moreover, a set of candidate objects that can still reach the top-k is maintained (Lines 12–13 in Algorithm 2). The algorithm returns results once none of the candidates can enter the top-k (Lines 15–16 in Algorithm 2).

In contrast to TA, NRA avoids expensive random accesses, however, it may need to scan large parts of the posting lists. Considering the taxonomy of top-k queries, both algorithms assume the top-k selection query model, they are exact methods computing over certain data using a monotone ranking function and are implemented at the application level. They only differ in the data access model, TA allowing sorted and random accesses, whereas NRA does not allow random accesses at all and relies completely on sorted accesses.

2.3 Top-k Rankings

The concept of rankings exists in all domains, providing information of the relative performance of items within a domain. Essentially, rankings allow focusing on a small subset of an exhaustively full list—usually a few top or bottom entries are of interest.

Formally, **rankings** are regarded as permutations over a fixed domain D and considered complete, that is, each item is contained in each ranking. A permutation ϕ is defined as a bijection from the domain $D = D_\phi$ onto the set

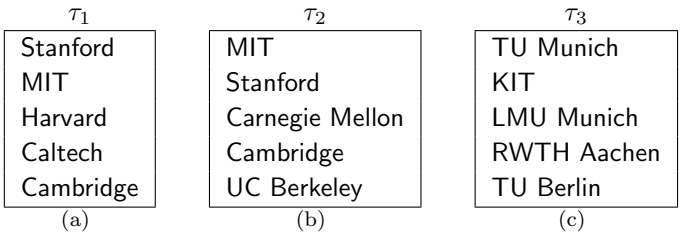


Figure 2.5: Sample rankings of top universities

$[n] = \{1, \ldots, n\}$ [FKS03]. The value $\phi(i)$ describes the rank of an element i in the permutation ϕ. An element i is said to be ranked higher of an element j in ϕ if $\phi(i) < \phi(j)$.

Incomplete rankings, called **top-k lists** by Fagin et al. [FKS03], present the k highest ranked entries in a complete ranking. Formally, a top-k list τ is a bijection from D_τ (the elements in the top-k list) onto $|k| = \{1, \ldots, k\}$. It is important to note that two individual top-k lists, e.g., τ_1 and τ_2, do not necessarily share the same domain, i.e., $D_{\tau_1} \neq D_{\tau_2}$. In this thesis, without any loss of generality, we assume that $\tau(i)$ takes values from 0 to $k-1$ (instead of 1 to k).

For instance, Figure 2.5 shows sample rankings containing (a) the top-5 universities in the world overall, (b) the top-5 universities for computer science in the world, and (c) top-5 universities for computer science in Germany. The position of MIT in τ_1 and τ_2 is $\tau_1(\mathsf{MIT}) = 1$ and $\tau_2(\mathsf{MIT}) = 0$, respectively. Note that τ_1 and τ_2 do not share the same domain, $D_{\tau_1} \neq D_{\tau_2}$, i.e., they contain different elements.

2.3.1 Indexing Rankings

Helmer and Moerkotte have shown in a study [HM03] that short sets, thus rankings, can very efficiently be indexed with traditional inverted indices. These indices, as described in Section 2.1.1 and known from being used in full-text search, keep for each item a list of rankings that contain the items. The inverted index as a structure allows for efficiently identifying those rankings that have one or more items in common with the query items. The key point of using inverted indices is their ability to efficiently reduce the global amount of all rankings to potential candidates by eliminating the rankings that do not contain the query items. The inverted index can be augmented to make it contain the rank information, hence, being able to check the positions of the items without looking-up the actual rankings. A posting in the index for an item i has the structure $(\tau : \tau(i))$, i.e., the ranking where it occurs τ and its position in that ranking $\tau(i)$.

A partial example positional inverted index following the sample rankings depicted in Figure 2.5 is shown in Figure 2.6. Looking up in the inverted index

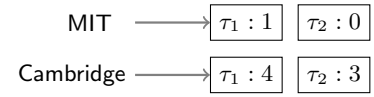


Figure 2.6: Sample positional inverted index of rankings

can reveal that MIT appears only in the rankings τ_1 and τ_2. Furthermore, the position of MIT in each of the rankings can also be retrieved.

2.3.2 Similarity Distance Measures

Data objects are frequently compared how much they are alike in different application scenarios. In order to quantify the resemblance between objects, various similarity measures were introduced. Depending on the application at hand, the data being binary or numerical, different measures are suitable comparators for evaluating the similarity between objects [LRB09]. In the context of similarity search, rankings and text documents are commonly considered as binary objects, both can be encoded as set data and described by the set of present characteristics. In comparing rankings, one of the most important distance measures are Spearman's Footrule distance [Spe94] and Kendall's Tau distance [Ken38], whereas the Jaccard coefficient [Jac12] is frequently used in comparing set data. Following, we discuss the Spearman's Footrule distance in more detail and describe some other distance functions considered in this thesis.

Spearman Footrule Distance

Spearman's Footrule distance is the $L1$ distance between two ranking lists. The rankings are regarded as permutations over a fixed domain D with n elements, as shown in Section 2.3. It can be considered as a measure of disarray between two permutations.

The Footrule distance between two rankings ϕ_1 and ϕ_2 is defined by:

$$F(\phi_1, \phi_2) = \sum_{i=1}^{n} |\phi_1(i) - \phi_2(i)|$$

It represents the absolute difference between the ranking positions of the elements in the ranking lists. The Footrule distance is a distance metric, that is, it has the symmetry property, i.e., $d(a, b) = d(b, a)$, is regular, i.e., $d(a, b) = 0$ iff $a = b$, and it satisfies the triangle inequality $d(a, c) \leq d(a, b) + d(b, c)$, for all a, b, c in the domain.

The minimum distance between two rankings is when they are identical and amounts to 0, whereas the maximum distance occurs when ϕ_1 is the reverse of ϕ_2 and is computed:

$$\max F(\phi_1, \phi_2) = \begin{cases} \frac{n^2}{2} & \text{when } n \text{ is even} \\ \frac{(n+1)(n-1)}{2} & \text{when } n \text{ is odd} \end{cases}$$

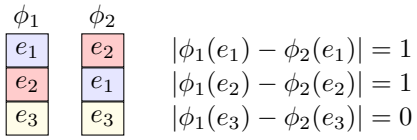


Figure 2.7: Spearman's Footrule distance

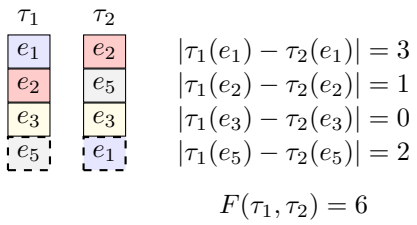


Figure 2.8: Generalized Spearman's Footrule distance

Knowing the minimum and maximum, the Footrule distance can be normalized, ranging from 0 to 1.

Figure 2.7 shows an example computation of the Footrule distance between two rankings. For instance, the position of e_1 in the rankings is $\phi_1(e_1) = 0$ and $\phi_2(e_1) = 1$, thus $|\phi_1(e_1) - \phi_2(e_1)| = 1$. The Footrule distance between the two lists is $F(\phi_1, \phi_2) = 2$.

Generalized Spearman's Footrule Distance. Top-k rankings are incomplete in nature, they capture the k most prominent elements in a domain. Hence, two top-k lists τ_1 and τ_2 do not necessarily share the same domain, i.e., $D_{\tau_1} \neq D_{\tau_2}$. The Footrule distance between top-k lists requires some adjustment, since simply avoiding the non-common elements in the computation can lead to information loss. Fagin et al. [FKS03] propose a generalized Spearman's Footrule distance that considers the incompleteness of top-k lists. The main idea is to consider that all missing elements from a certain top-k lists are occurring after the last element of the list, i.e., at position $k + 1$.

The Footrule distance between two top-k rankings τ_1 and τ_2 is defined as follows:

$$F(\tau_1, \tau_2) = \sum_{i=1}^{n} |\tau_1(i) - \tau_2(i)|$$

where $\tau_2(i) = k + 1$ if $i \in \tau_1$ but $i \notin \tau_2$, and vice-versa.

The minimum distance between two top-k lists is when they are identical and amounts to 0, while the maximum is when two disjoint lists are compared and is equal to:

$$\max F(\tau_1, \tau_2) = k(k + 1)$$

Figure 2.8 shows an example computation of the Footrule distance between

two top-k lists τ_1 and τ_2. Note that, for instance, $e_1 \in \tau_1$ but $e_1 \notin \tau_2$, thus in computing the distance $\tau_2(e_1) = k + 1$.

Jaccard Coefficient

The *Jaccard coefficient* [Jac12] measures the similarity and diversity between two sets by looking at how many common elements they share. The Jaccard coefficient between two sets A and B is defined as the size of their intersection over the size their union:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Jaccard is a symmetric distance measure, i.e., $J(A, B) = J(B, A)$. For disjoint sets, $J(A, B) = 0$. It can be computed for an arbitrary number of sets as follows:

$$J(A_1, \ldots, A_n) = \frac{|\bigcap_{i=1}^{n} A_i|}{|\bigcup_{i=1}^{n} A_i|}$$

Numerical Distance Measures

For numerical data, the most common distance measure is the *Euclidean distance*, also known as the $L2$ distance. The Euclidean distance between two data objects $A(a_1, \ldots, a_n)$ and $B(b_1, \ldots, b_n)$ is defined as:

$$d(A, B) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

Another prominent distance metric is the *Manhattan distance*, also known as $L1$ distance. It is the absolute difference between the points in data objects. The Manhattan distance between two data objects $A(a_1, \ldots, a_n)$ and $B(b_1, \ldots, b_n)$ is computed with:

$$d(A, B) = \sum_{i=1}^{n} |a_i - b_i|$$

Both distances can be normalized in the interval $[0, 1]$ using the min-max normalization:

$$d_n = \frac{d - d_{min}}{d_{max} - d_{min}}$$

where d_n is the normalized distance, d the actual observed distance, and d_{min} and d_{max} respectively are the minimum and maximum observed distances in the entire dataset.

2.4 Online Analytical Processing (OLAP)

Decision making in organizations increasingly relies on comprehensively analyzing, exploring, and discovering trends in current and historical data [RG03]. In

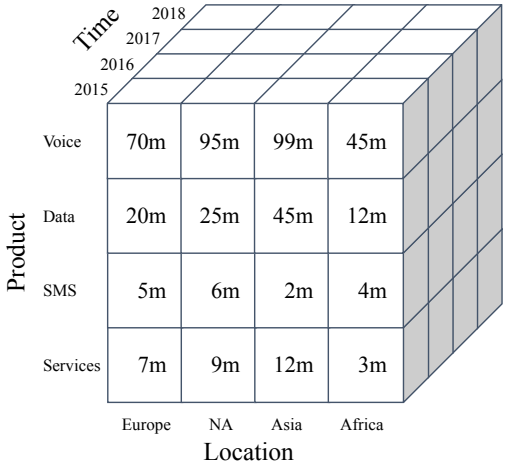


Figure 2.9: A data cube

effort of creating such **decision support systems**, traditional database systems were augmented with new constructs and techniques, enabling support of complex queries. These queries typically include a group-by clause, aggregation and statistical functions, support for complex boolean conditions, and features for time series. Applications designed for such queries are labeled *online analytical processing (OLAP)*. Such systems represent data as a multidimensional array and are designed towards end-user tools, e.g., spreadsheets. Data from several different databases is consolidated into one central repository called a *data warehouse*. A data warehouse stores summaries of recent and past data that can be correlated and analyzed for better business intelligence.

2.4.1 Data Model

Data in OLAP applications is represented over multiple attributes, also called dimensions, with the focus on a collection of numeric measures. For instance, a telecommunications operator that is interested in the revenue from their products in different time periods and locations, can organize their data over the attributes product, location, and time. On the other hand, revenue represents the measure attribute and a single instance of revenue is identified as a combination of the different dimension values.

The data organization of attributes into dimensions and numeric measures is typically represented with a *data cube*. Figure 2.9 shows a sample data cube storing data about the previous example. A cell in the data cube represents an aggregated value of the measure of interest revenue, which is summarized over each of the dimension attributes product, location, and time.

The values in each dimension attribute can be organized as a hierarchy, if such an opportunity exists. In the example, the time attribute, can be organized into weeks, months, quartiles, and years. Then, the data cube will contain an aggregated value for each time period in the hierarchy.

tid	Items
1	{bread, milk}
2	{bread, diapers, beer, eggs}
3	{milk, diapers, beer}
4	{bread, milk, diapers, beer}
5	{bread, milk, diapers}

Table 2.2: Sample market basket transactions database [TSK05]

Storing the dimensions and measure attribute in a database is commonly done in different tables. The table containing the measure attribute is referenced as the *fact table* and is the central model table in the data warehouse. There are two models in organizing data in data warehouses using relational databases: *star schema* and *snowflake schema* [RG03]. Both models contain the fact table at the center, surrounded by *dimension tables* referenced using a primary-foreign key relationship. In a star schema each dimension is denormalized into one table, whereas in the snowflake schema some of the dimensions are normalized—those that are structured in a hierarchy.

2.5 Knowledge Discovery

Interesting or unexpected trends and patterns in large unstructured data can be discovered with exploratory data analysis. The challenge of developing methods for extracting knowledge from large amounts of data is the focus of the broad field of *knowledge discovery and data mining* [TSK05]. A wide range of techniques can be used to support a variety of business applications, such as targeted marketing, customer profiling, store layout, and fraud detection. Depending of the application and problem at hand, different methods are utilized. We briefly consider the subjects of frequent itemset mining and the representation of classification rules with decision trees.

2.5.1 Frequent Itemset Mining

Data collection about customer purchases has allowed retailers opportunities to develop analytics techniques such as market basket analysis to analyze their shopping behavior. A *market basket* is a collection of items bought by a customer in a single customer transaction, for instance a single order from an online retailer. Retailers are interested in learning which items are purchased together. Such valuable information can be used in targeted marketing promotions or improving the layout of items in a store. Table 2.2 shows an example of market basket transactions. Each customer transaction represents a row in the table and it contains the set of items bought in a single purchase.

An *itemset* is a collection of items. Formally, a transaction $t = (tid, I)$ is an itemset I with associated transaction identifier *tid*. A transaction t contains an

method: **aprioriAlgorithm**

```
1  k = 1
2  /* find all frequent itemset of size 1 /*
3  for each item i ∈ D
4    if freq(i) > minfreq then add i to F_k
5  /* compute score(d) */
6  repeat
7    k = k + 1
8    /* generate candidate itemsets */
9    C_k = apriori-gen(F_{k-1})
10   for each transaction t ∈ D
11     /* identify all candidates that belong to t */
12     C_t = subset(C_k, t)
13     for each candidate itemset c ∈ C_t
14       /* increment support count */
15       supp(c) = supp(c) + 1
16   /* extract the frequent k-itemsets */
17   for each candidate c ∈ C_k
18     if freq(c) > minfreq then add i to F_k
19 until F_k = ∅
20 return ⋃ F_k
```

Algorithm 3: Apriori algorithm

itemset A if $A \subseteq I$. The *support of an itemset* A in a database D is the number of transactions $t \in D$ that contain A:

$$supp(A, D) = |\{t \in D : t \text{ contains } A\}|$$

The *frequency of an itemset* A in a database D is the fraction of transactions in the database that contain the itemset:

$$freq(A, D) = supp(A, D)/|D|$$

An itemset whose frequency is above a user-specified threshold of minimum frequency $minfreq$, is called a *frequent itemset*.

For instance, in the sample transactions database shown in Table 2.2, the itemset $A = \{\text{beer}, \text{milk}\}$ has support $supp(A, D) = 3$ and frequency $freq(A, D) = 3/5$. With a support threshold $minfreq = 1/2$, it is a frequent itemset.

Apriori Principle

The apriori principle and algorithm [AS94] are a widely known approach to frequent itemset mining.

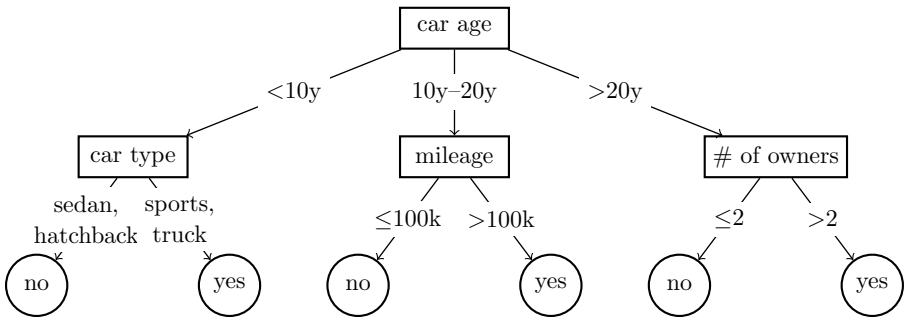


Figure 2.10: Sample decision tree

Theorem 2.5.1 Apriori Principle
If A is a frequent itemset, then all of its subsets $B \subseteq A$ must be also frequent.

For example, if {beer, milk} is a frequent itemset, then {beer} and {milk} are also frequent. Conversely, if {beer, milk} is infrequent, then all supersets of {beer, milk} are infrequent, too. The apriori principle can reduce the number of itemsets that need to be examined by discarding supersets of an already infrequent itemset. The strategy of reducing the search space based on the support measure is known as *support-based pruning*.

Definition 1 Monotonicity Property
Let I be a set of items and $J = 2^I$ be the power set of I. A measure f is ***monotone*** *(or* ***upward closed****) if:*

$$\forall A, B \in J : (A \subseteq B) \rightarrow f(A) \leq f(B)$$

Definition 2 Anti-monotonicity Property
Let I be a set of items and $J = 2^I$ be the power set of I. A measure f is ***anti-monotone*** *(or* ***downward closed****) if:*

$$\forall A, B \in J : (A \subseteq B) \rightarrow f(B) \leq f(A)$$

The support measure has the anti-monotone property. The apriori approach uses the anti-monotonicity of the support to significantly reduce the search space. Algorithm 3 describes the details of the apriori algorithm.

2.5.2 Decision Trees

Classification has a wide range of applications, such as predicting tumor cells as benign or malignant, classifying credit card transactions as legitimate or fraudulent, and categorizing news stories [TSK05]. In a classification task, the input is a collection of training records, each containing a set of attributes. One distinguished attribute is called a class, which is a categorical attribute with a small domain. The goal is to build a concise model as a function of the values

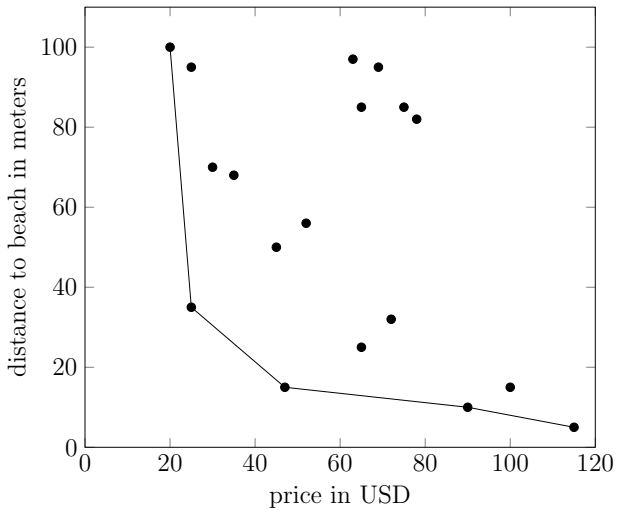


Figure 2.11: Sample skyline of hotels

of the other attributes. This model should then assign a class to a record as accurately as possible.

A *decision tree* is a tree-like structure that can be used to represent classification rules. It is a directed, acyclic graph, where the root node does not have incoming edges, and every other node has zero or more outgoing edges and exactly one incoming edge. Each internal node in the tree contains attribute test conditions to separate records with different characteristics; each leaf node is labeled with one class label.

A simple decision tree representing car insurance risk classification is depicted in Figure 2.10. Classification rules are defined with each path from the root node to a leaf node. For instance, if a car is older than 20 years and has more than 2 owners it has a high insurance risk.

2.6 Skyline Query Processing

The concept of *skyline query processing* in databases was first introduced by Börzsöny et al. [BKS01] but its core principle is also known as the maximum vector set problem [KLP75] or Pareto optimality.

Given a d-dimensional dataset D, a skyline query returns as result all data objects which are not dominated by any other object in D.

Definition 3 Dominance in Skyline Processing
A data object $A(A_1, \ldots, A_d)$ *dominates another object* $B(B_1, \ldots, B_d)$ *if:*

(1) For every dimension $i \in \{1, \ldots, d\} : A_i \leq B_i$*; and*

(2) For at least one dimension $j \in \{1, \ldots, d\} : A_i < B_i$*.*

The dominance relationship has the transitive property, i.e., if A dominates B and B dominates C, then A dominates C.

A common and intuitive example to underpin the idea behind skylines is the search for hotels that are ideally both close to the beach and cheap. A hotel that costs \$125 a night and is only 75m away from the beach would dominate a hotel that is more expensive and farther away from the beach. The latter would be out of question then (not in the skyline). But a hotel that is 50m away from the beach although it costs \$130 a night would not be dominated by the first and, thus, be part of the skyline result. Figure 2.11 shows the skyline of cheap hotels close to the beach for a sample dataset containing hotels.

Various skyline query processing algorithms, using different approaches, have been proposed by the research community. Börzsöny et al. [BKS01] proposed BNL, D&C, and an algorithm using B-trees. Other prominent methods include the Bitmap algorithm [TEO01], the NN algorithm [KRR02], the BBS method [PTFS05], and SFS [CGGL03].

The traditional skyline model is later extended and different variants are introduced, as discussed in the related work chapter with respect to applicability to our context.

Chapter 3

Related Work

This chapter gives an overview of the state-of-the-art approaches in extracting knowledge and understanding relationships in data with different data exploration techniques. First, in Section 3.1, we describe various solutions to reverse engineering database queries—techniques aiming to assist users in discovering data characteristics through the information provided in the identified queries, each of them focusing on different query and/or input types. Furthermore, Section 3.2 presents existing data exploration systems that consider obtaining knowledge from potentially unfamiliar data. As we deal with dominance between entities, literature on skyline processing is naturally highly relevant. Section 3.3 describes variations of skyline query processing with their relevance and differences to finding dominant entities in rankings.

3.1 Reverse Engineering Database Queries

There are a variety of approaches in the research community that resort to use reverse engineering queries as a proxy for exploratory data analysis (cf., [MLVP17] for an overview). They consider specific or approximate tabular input and usually focus on discovering certain query types. Most of the existing approaches consider input in form of a table and a subclass of select-project-join SQL queries, without arithmetic expressions [TCP09, SPGW10, ZEPS13, KLS18]. Techniques are also developed that focus on dealing with group by and aggregation [TCP14, TZES17], or are designed to handle textual input containing few example tuples [SCC$^+$14, PDCC15].

The problem of reverse engineering queries was considered by Tran et al. [TCP09] in their data-driven approach called TALOS. Given an input query Q, a database D, and a query output $Q(D)$ produced by the query Q, they try to find an instance-equivalent query Q'. They define *instance-equivalent queries* as those that produce the same output. The projection and join predicates are identified using the input query Q and the focus is on identifying the selection predicates in select-project-join queries. Generating the selection conditions is

defined as a data classification task and they use a decision tree classifier that is constructed in a top-down manner in a greedy fashion by determining a fitting predicate, based on the Gini index, according to which the tuples are split into two classes. These two classes would then form the root nodes of two decision trees (constructed recursively). In later work [TCP14], they generalize their approach into not knowing the input query Q, handling more expressive queries containing group by and aggregations, and dealing with multiple versions of the database D. In discovering the join relations, domain indices are used which are defined on all attributes in the database that share the same domain. Furthermore, they present heuristics and use standard pseudo-polynomial algorithms [CLRS01] to discover the aggregation function in the query, and adjust their approach to consider having multiple versions of the database as input by using well-known join view maintenance techniques [BLT86, MQM97].

Following our work, the problem of reverse engineering OLAP queries was also considered by Tan et al. [TZES17] in which they propose a three-phase algorithm named REGAL. They consider a single base table R and a query output table $Q(R)$ which is generated from R using an aggregation query. The focus is on discovering the group by and aggregation components of the query in presence of possible selection conditions and they do not consider joins. In the first phase, they propose a lattice exploration strategy that can discover group-by candidates by pruning the invalid candidates in the lattice. The lattice is constructed by using the maximum set of query columns that can be potential group-by attributes as a root node and subsequently creating the remaining nodes according to the superset-subset relationship. In order to identify the group-by candidates, keyness and containment checks are performed. For each of the grouping columns, aggregate columns and functions are identified in the next step. They present a set of table-level, group-level, and multi-column aggregation constraints for quickly invalidating incorrect combinations of aggregations. In the last step of their approach, they propose an algorithm to find the selection conditions by looking for fuzzy bounding boxes in multi-dimensional matrices, with dimensions based on the attributes in the base table. The selection conditions considered are conjunctions of range or equality predicates. A cross-validation strategy across multiple matrices, each corresponding to a specific group and a specific aggregation function, is used to prune the invalid bounding boxes until the predicate is identified. This work is the most similar to the query type supported in this thesis, however it differs in that the input in our work is a ranked top-k list.

Sarma et al. [SPGW10] explore a related problem called the *View Definitions Problem* (VDP) which identifies a view definition Q given an input database D and a materialized view V. This work considers a query class with only one relation R and there are no joins or projections. The focus is on analyzing the complexity of different variants of the VDP problem for diverse query variations in terms of the number of equality or range predicates, and whether conjunctive or disjunctive predicates are allowed. Thus, they only try to find the selection condition of Q and do this looking at the problem as an instance of the set cover

problem [Vaz01]. In addition, they study the level of approximation of the result query Q' that may not exactly match the original query Q that defined the view V. As a difference metric for the approximate queries they use the symmetric set difference $d(V, Q(R))$. From the families of queries that they cover, we focus on conjunctive queries with a single equality predicate and conjunctive queries with any number of equality predicates. For both types they propose naive algorithms that utilize the size of the attribute domains in the view.

Reverse engineering complex join queries is studied by Zhang et al. [ZEPS13]. Their main insight is that any graph can be characterized as a union of disjoint paths, connecting its projection tables to a center table, called a *star*, as well as a series of merge steps over this star table. Using this insight, the proposed algorithm filters out candidate queries that need not be tested. To keep track of the candidates, a lattice structure is used where each vertex represents a star and the edges represent merge steps. While this system allows to efficiently discover complex join queries, it is not able to handle top-k OLAP queries containing aggregations or selections. We use their approach as guidance in discovering the join predicates, however using a small top-k list as input and supporting aggregation functions change the problem significantly.

Kalashnikov et al. [KLS18] propose a more efficient approach in discovering complex join queries, named FastQRE. Moreover, they support and define a broad class of covering project-join (CPJ) queries based on covering conditions defined on the query graph. The FastQRE framework consists of four modules. First, pre-processing is performed on the input data, with parsing, computing column-cover, and building database indices. Then, a candidate query generation module tries to generate a good sequence of candidate queries, by identifying correct column mapping and identifying correct join paths using a standard breadth-first search algorithm for discovering walks in the schema graph. The candidate queries need to be validated on the database to check if they produce the input relation. In order to avoid expensive executions of complex queries on a potentially large database, the system tries to dismiss wrong queries by employing techniques such as probing queries trying to discover certain discrepancies and checking for walk coherence. Finally, a progressive query evaluation on the database is performed, which gets results one tuple at a time and propagates useful information feedback to the candidate query generation module.

Shen et al. [SCC+14] study the problem of discovering a minimal project-join query that contains the exact set of given example tuples in its output and do not consider selections. The focus is more on the verification of candidate queries. They require the database tables to contain only text columns with keyword search allowed on them and introduce a candidate generation-verification framework to discover all valid queries. By using common sub-join trees of the candidate queries as filters, which rely on full-text-search indices on each text column in the database, they manage to improve the efficiency of their approach.

Psallidas et al. [PDCC15] propose a candidate enumeration and evaluation

framework with flexible caching component for discovering project-join queries which approximately contain a given set of example tuples. Their system handles only text columns, provides a spreadsheet style keyword search system, and relies on offline creation of large indices on the text columns. The system returns the project-join queries with the top-k highest scores. Their scoring model allows to tolerate relationship and domain errors with the given example tuples to make up for human errors while still providing a relevant top-k list of queries. Moreover, they propose a caching-evaluation scheduler, where they dynamically cache common sub-expressions that are shared among the PJ queries.

None of these systems allow reverse engineering of complex top-k OLAP queries including aggregations and foreign/primary key joins based on a ranked top-k input.

Another approach to reverse engineering queries is where users also specify negative example tuples that the output should not contain. Bonifati et al. [BCS14, BCS16] consider discovering an n-ary join predicate across m relations by users interactively labeling a set of candidate tuples as positive or negative examples, depending whether or not they should be part of the join result. The focus is on inferring joins with equality predicates and disjunctive join predicates. The presented approach prunes the search space by trying to present suitable tuples to the user with the goal of minimizing the user interactions. Weiss and Cohen [WC17] examine the computational complexity of discovering SPJ queries from positive and negative example tuples. They provide a theoretical study where several classes of queries are considered, with different combinations in the operators. Furthermore, the size of the query is also considered, as well as the size of the schema and the number of examples as input.

Wang et al. [WCB17] describe an approach for reverse engineering SQL queries on small databases with fewer than 100 cells. Their approach enumerates abstract SQL queries that can satisfy the given example tuples and then looks for actual predicates for each abstract query. In order to improve efficiency, predicates are clustered into semantically equivalence classes and tables are encoded using bit-vectors. However, this enumeration technique does not scale to large databases.

Li et al. [LCM15] propose finding queries from examples, to help non-expert database users construct SQL queries. The user first provides a sample database D and an output table R which is the result of Q on D. The proposed approach is able to identify the user's target query by asking for the user's feedback on a sequence of a slightly modified database-result pairs, which are generated by the system. They use the work of [TCP14] for generating candidate queries and focus on optimizing the user-feedback interactions to minimize the user's effort in identifying the desired query.

The principle of reverse query processing is studied by Binnig et al. [BKL07], however their objectives and techniques are different. They consider generating a database instance D such that given a query Q, the schema of a relational

database, and a desired result R, executing Q on D would result in R. They combine techniques from traditional query processing and model checking and propose a reverse relational algebra, where each relational algebra operator has a corresponding operator implementing its reverse function. The focus is to generate minimal size test databases for functional testing of database applications. The QAGen system [BKLÖ07] introduces an approach focused on DBMS testing that generates a test database given a test query, by introducing symbolic query processing which relies on constraint solvers to generate the database. Other research efforts [BCT06, MKZ08] study the problem of generating query parameters for test queries to meet certain cardinality constraints on their subexpressions, with the test databases given.

A reverse top-k query [VDKN10] returns for a point q and a positive integer k, the set of linear preference functions (in terms of weighting vectors) for which q is contained in their top-k result. For example, finding all customers who treat the given query product q as one of their top-k favorite elements. In such cases, each customer is described as a vector of weights. Although it appears related given the name, this research area is not directly related to our work.

3.2 Data Exploration Techniques

Research in data exploration considers efficiently extracting knowledge from potentially unfamiliar data. The constant increase of amounts of data requires constructing more dynamic data-driven exploratory applications that help users effortlessly discover information relevant to their interest. Idreos et al. [IPC15] provide an overview of past and emerging research in the area.

An approach, named YMALDB, for result-driven recommendation is presented by Drosou and Pitoura [DP13] which proposes additional items to the users that are highly correlated to the result of their original query. In this way, users are exploratory steered towards parts of the database that they would not have seen with their original query. Interesting parts of the result, called faSets, are identified based on their frequency in the result and the database, with exploratory queries based on the interesting faSets are further constructed and their results presented to the user. The efficiency of the proposed approach is based on estimating the frequency of the faSets by maintaining a set of representative closed rare faSets. The size of the set that is maintained determines the estimation accuracy which is parameter-tunable. The algorithm runs in two phases, first it uses the pre-computed summaries to set a frequency threshold, which is then used to run a frequent itemset-based algorithm on the result of the query.

In [DPD14], Dimitriadou et al. introduce the AIDE framework that iteratively steers users towards interesting data areas by leveraging relevance feedback on data samples. It combines classification algorithms and data management optimization techniques to automate the data exploration tasks and delivers query predictions for common conjunctive and disjunctive queries. The user

is asked to label a set of sample objects as relevant or irrelevant to their task and based on this feedback the system generates a profile that is used to collect a new set of sample objects. The profile is utilized to generate data extraction queries that return more relevant than irrelevant objects to the user. In [DPD16] they propose optimizations to improve the efficiency of the system by reducing the exploration overhead. A skew-aware exploration technique is described, which deals with user interests in skewed data spaces. Furthermore, they introduce a probabilistic sampling strategy for presenting suitable samples to the user and extend the relevance feedback model to not only relevant/irrelevant samples, but also allowing annotation of similar samples.

Khan et al. [KSA14] describe a system for efficient result diversification in interactive data exploration. Their schemes leverage the overlap between query results across exploratory sessions and utilize a model-based approach using a cache of diversified results for improving the efficiency of diversification. This technique can be combined with the relevance score from other exploration methods to provide a more diversified set of samples to the user.

Sidirourgos et al. [SKB11] propose SciBORQ, a system that supports scientific exploration queries with control over quality and runtime of query execution. The approach relies on large samples, named impressions, biased towards the query workload and uses adaptive based sampling to adapt to shifting query workloads. It is a multi-layer data exploration framework utilizing bounded query processing with control over the runtime, disk space, and statistical quality.

Sellam and Kersten [SK13] present Charles, a query advisory system designed to query the query space associated with a given database. Their idea is to infer queries from queries using a proprietary query language that breaks the extent of a query into meaningful segments formed by conjunctive predicates. In this way, the user gets insight into the dataset and gets directions for further exploration. The segmentations are generated using a heuristic that splits the data into two equal parts recursively on dependent attributes until a threshold is reached.

Jiang and Nandi [JN15] introduce a snapping technique that guides users through the query space by providing interactive feedback during exploratory query specification. The recommended queries can be derived from the data or from prior query logs. They consider different feedback mechanisms at the user and device level and propose a data-driven approach with interactive times, considering data reduction strategies that reduce the dataset, but still maintain its characteristics.

Kalinin et al. [KÇZ14] describe an approach based on window-queries where users can specify shape- and content-based predicates to interactively explore a database. The framework provides interactive response time by relying on incremental online processing and integrating stratified sampling, adaptive prefetching, and data placement techniques. They structure the search space as a graph, define relationships between windows, and try to dynamically generate promis-

ing candidate windows to be explored in a suitable order. Furthermore, they present a diversification approach that directs the framework to unexplored regions of the search space.

Searchlight [KÇZ15] is a system that enables exploration of multidimensional data based on constraint programming [RvBW06] integrated into a DBMS. It incorporates APIs for constraint solvers as parts of query plans and introduces optimization algorithms for parallelization. The technique first speculative executes the query using a constraint-programming solver over a synopsis which produce candidate solutions. The candidate solutions might contain false positives, therefore are later validated over the original data. The system supports distributed processing and uses data- and search-space balancing techniques.

Joglekar et al. [JGP15, JGP16] present a data exploration system employing a novel smart drill-down operator that can be used to discover and summarize interesting groups of tuples. The tuple groups are each described by rules (value patterns) which should capture the interesting aspects of the table. For instance, the rule $(\mathsf{IKEA}, \mathsf{chairs}, \star, 600)$ describes that there are 100 tuples with IKEA as the first column and chairs as the second, which can mean to the user that IKEA is selling a lot of chairs. Furthermore, they present an algorithm for discovering the approximately optimal list of rules to summarize the data and propose a dynamic sampling scheme for handling large data.

A prominent data exploration technique is faceted search (e.g., [KJTN14, KHP10]) where query results are classified into categories (facets), and the user refines them by selecting one or multiple facets. In structured datasets, some of the research challenges include the question which facets should be displayed to the user and which facet conditions can lead to the desired items of interest in less navigation/interaction steps.

DICE [KJTN14] supports efficient exploration of data cubes using faceted search by utilizing pre-computed samples and a distributed DBMS. To achieve interactive response time, the framework relies on a cost-model that combines speculative query execution and online data sampling in a distributed setting with data distributed as table shards across nodes. A user can explore a cube by examining a facet of certain portion of the data cube and can then traverse the cube from that facet to another. The successive facets reflect data cube operations, i.e., a parent facet corresponds a rollup, a child facet reflects a drilldown, a sibling facet signifies a change of a dimension value in the group, and a pivot facet is obtained by a change in the inspected dimension. Thus, using these traversals, the user can fully explore the cube.

Kashyap et al. [KHP10] introduce FACeTOR, a cost-driven exploratory navigation, based on a cost model for user actions, approximated using user relevance judgments over attributes. During the navigation, the system shows a subset of possible facet conditions that are chosen to minimize the overall expected navigation cost. To estimate the navigation cost they introduce a probabilistic model that models the user interactions and present heuristics that utilize the cost model to efficiently compute the optimal set of suggested facets. They

propose a method inspired by the approximation algorithm for the weighted set cover problem [Chv79] that tries to identify a relatively small set of suggestions that are highly recognizable for the users, as well as a method following their cost model.

Research in schema mapping, specification of the relationship between a source schema and a target schema, is also related to our work. Miller et al. [MHH00] present a framework for semi-automated schema mapping where given a set of specifications for transforming values from input tables into target tables, it discovers the most likely mapping SQL queries needed for the transformations. Alexe et al. [AtCKT11] describe a system where the user specifies example tuples and the system suggests transformation rules which can then be edited by the user. Gottlob and Senellart [GS10] introduce a theoretical framework and provide a systematic analysis of the complexity for automatic discovery of relational schema mappings based on example data instances.

In keyword search over databases [ACD02], the input is a single tuple with specified keywords as fields. The works of [BJK+12, TL08] interpret the query intent behind the keywords and compute aggregate SQL queries. Blunschi et al. [BJK+12] use patterns that interpret and exploit different kinds of metadata, while Tata et al. [TL08] discover aggregate SQL expressions that describe the intended semantics of the keyword.

All these systems are adjacent to the work in this thesis, but share the same goal—all are focused on facilitating knowledge extraction from data in an exploratory manner.

3.3 Dominance in Skyline Query Processing

The concept of skyline query processing in databases was first introduced in [BKS01]—we described its traditional notion of dominance in more detail in Section 2.6. To recap, given a dataspace D with d-dimensional objects $A(A_1, \ldots, A_d)$, the skyline consists of all data objects which are not dominated by any other object in D. An object A dominates another object B if it is better than or equal to B in all dimensions and better than B in at least one dimension. The research community extended the traditional skyline model and various forms of skyline processing are proposed in literature.

Papadias et al. [PTFS03] introduced different variants of the skyline operator, such as subspace, constrained, and dynamic skyline queries. A subspace skyline query refers only to a user-defined subset of dimensions U. The subset skyline are the objects in D which are not dominated by any other object on the dimensions in U. In a constrained skyline query, each constraint is usually expressed as a range across a dimension, with the conjunction of all such constraints forming a hyper-rectangle in the d-dimensional attribute space. The constrained skyline is a subset of the skyline objects that satisfy the given constraints. Finally, they propose dynamic skyline queries which consider dynamic attributes of data objects to be computed by a set of dimension functions. A

dynamic skyline query specifies m dimension functions $f_1, \ldots, f_m$ such that each function f_i takes as parameters a subset of the coordinates of the data points and returns the skyline in the new data space with dimensions defined by $f_1, \ldots, f_m$.

Chen and Lian [CL08] define more generic dynamic skyline queries in metric spaces. Given n query objects $q_1, \ldots, q_n$, a metric skyline query computes the data objects in D whose attribute vectors are not dominated by those of other data objects, where the attribute vector of an object is defined as a n-dimensional vector consisting of metric distances from this object to n query points. Moreover, they present an algorithm using triangle-based pruning that applies the triangle-inequality in metric spaces and use an M-tree index [CPZ97] to compute the metric skyline points. However, they consider complete data across all dimensions, which differs from our problem in that our entities do not occur in all the rankings, and, moreover, dynamic skyline queries still look for the skyline using the dynamic coordinates.

With large number of dimensions, the chance of an object dominating another object is very low. To this purpose, Chan et al. [CJT+06] introduce k-dominant skylines and k-dominance. An object A k-dominates another object B if there are k ($\leq d$) dimensions in which A is better or equal than B and there is at least one of these dimensions where A is better. The object that is not k-dominated by any other in the dataset is in the k-dominant skyline. Thus, the conventional skyline is a special case of the k-dominant skyline, where $k = d$. The transitive property of the dominance relationship in skylines does not hold for k-dominance. The proposed algorithms are adjustments to BNL [BKS01] and a rank aggregation algorithm [FLN01] that pre-sorts objects separately according to each dimension and merges the ranked lists.

Mindolin and Chomicki [MC11] define prioritized skyline (p-skyline) queries as a generalization of skyline queries where the user can prioritize some of the attributes by specifying them as more important with respect to the syntax of so-called p-expressions. Thus, the skyline semantic is enhanced with the notion of attribute importance. Given two attributes A_1 and A_2 such that A_1 is more important than A_2, an object with a better value in A_1 is always preferred to all objects with worse values in A_1, without taking into account the values in A_2. However, if two objects have the same value in A_1, then the one with better value in A_2 is preferred. For instance, given a dataset of cars characterized by age and price, some people prioritize price and consider age as a secondary criterion, which will be taken into account only when the price of two cars is the same. In order to solicit attribute importance, they develop a method of elicitation of attribute priority based on user-provided feedback on desirable and undesirable object examples.

Ciaccia and Martinenghi [CM17] introduce the notion of restricted skyline (or R-skyline), which takes into account the different importance of object's attributes. The R-skyline combines the approaches of top-k queries and traditional skylines, by modeling the priority of attributes through arbitrary constraints on the space of the weights in the scoring function used to rank the skyline objects.

Furthermore, they define the concept of $\mathcal{F}$-dominance, in which, given a set of monotone scoring functions $\mathcal{F}$, an object A dominates another object B if A is always better than or equal to B according to all the scoring functions in $\mathcal{F}$, and strictly better for at least one scoring function in $\mathcal{F}$. They focus on discovering the set of all non-$\mathcal{F}$-dominated objects and the top-1 (optimal) object according to some function in $\mathcal{F}$, with approaches based on linear programming.

Groz and Milo [GM15] study the problem of computing skyline queries where the input data objects can only be compared through noisy comparisons. Motivated by crowdsourcing scenarios, in this setup they study how to compute the skyline by asking users if object A is better than object B according to some attribute. The objects in their model are fully ordered along each dimension, with the order unknown but objects can be compared through oracles, i.e., asking a new user to compare two objects on a particular dimension. In such setting, the users are likely to make mistakes with some probability and confidence can be increased by performing independent repetitions of a comparison. Thus, they develop a model that describes how many comparisons are required to return the correct skyline with a high probability. Furthermore, they introduce algorithms that rely on sorting, binary search, and maxima procedures to compute skyline queries in the noisy comparison model.

Khalefa et al. [KML08] investigate skylines over incomplete data. With incomplete data it is assumed that not all dimensions are available for all data items, i.e., each object in the dataset $A = (A_1, \ldots, A_d)$ has at least one known dimension A_i, while all other dimensions have a non-zero probability of being unknown. Furthermore, for two objects A and B that may have incomplete dimensions, A is said to dominate B if (1) there is at least one dimension i where both A_i and B_i are known and A_i is better than B_i, and (2) for all other dimensions $j, j \neq i$, either A_j is unknown, B_j is unknown, or A_j is better or equal than B_j. Conversely, when comparing two incomplete objects A and B, only the common dimensions that are known for both points are considered. The skyline relation in incomplete data is non-transitive and may be cyclic for a given dataset of objects. Moreover, they propose an incremental algorithm that reduces the number of comparisons required by using virtual points and so called shadow skylines. Zhang et al. [ZLOT10] propose to convert a given incomplete dataset into a complete one by inserting estimated values of each missing dimension, based on the probability distribution function induced by the non-missing values in that dimension. In this way, they define a mapping dominance relationship over data with missing values which retains the transitivity property, making it suitable for using existing algorithms from traditional skyline query processing.

Probabilistic skyline queries on uncertain data were introduced by Pei et al. [PJLY07], where the probability of an item being in the skyline is the one that the item is not dominated by any other item. The so called p-skyline is then defined as the set of items where each one has a probability of at least p of being in the skyline. They focus on the discrete case of probabilistic skyline computation, where each uncertain object is represented by a set of instances. A pro-

posed bottom-up algorithm computes the skyline probabilities of some instances of the uncertain objects, which are used as bounds to prune other instances of uncertain objects. Moreover, a top-down algorithm partitions the instances of uncertain objects using partition trees into subsets recursively, bounds the skyline probabilities of nodes in the tree, and uses the bounds to prune parts of the tree. Böhm et al. [BFO+09] consider the case when objects are initially represented by probability density functions and propose methods that exploit index structures to efficiently compute the skyline. Zhang et al. [ZGH+17] extend the probabilistic skyline to incomplete data by describing each missing dimension value by the probability density function of the dimension. Furthermore, they propose an algorithm that uses pruning strategies, relying on bitmaps and bounds, to reduce the number of objects for which skyline probabilities are computed and implement optimizations with buckets and early termination technique comparing bounds of probabilities.

A top-k dominating skyline query [PTFS05, YM07, TPM11] returns the k objects with highest domination power, defined as the number of items that an object dominates. Thus, it combines the benefits of top-k and skyline queries in that no specialized scoring functions are required and the output size can be controlled. In our approach for computing dominant entities in rankings we define the most dominant entities differently, as those that dominate the input entities in the largest number of rankings across the dataset.

Skyline queries compute all objects that are not (probabilistically) dominated by any other object with regards to the entire dataset, which differs from our goal. In computing dominance in rankings, the reference point are the entities in the input, together with the rankings where they appear and the resulting dominating entities dominate the input, while they still can be dominated by other entities in the dataset.

To the best of our knowledge, there has not been any work that considers dominance of ranked entities for investigating data.

Chapter 4

Reverse Engineering Top-k Database Queries

4.1 Introduction

This chapter presents a detailed description of PALEO—our approach to reverse engineering top-k database queries. It is based on our own publications at EDBT 2016 [PM16] and KEYS 2016 [PMM16]. The name PALEO comes from being approximately the reverse of the word OLAP and also emphasizes the goal of assembling queries based on their data footprints (results), much like paleontologists reconstruct and study fossils. Given an input result list, PALEO is able to efficiently determine input-generating SQL queries and can additionally be relaxed in order to find queries that generate rankings similar to the input within a certain distance bound.

Reverse engineering database queries describes the task of obtaining an SQL query that is able to generate a specified input table, when executed over a given database instance. This generic problem has various important application scenarios as described in Chapter 1, specifically for top-k database queries that often yield valuable analytical insights. Consider, for instance, business analysts who are interested in determining alternative queries that yield the same or similar query result tuples, data scientists who try to find explanatory SQL queries for crowd-sourced top-k rankings, or to find the data-generating query of a sample input in order to re-execute it on current or future database instances in cases where the original query has not been saved or has not been made public, for one or another reason. The discovered queries can reveal interesting properties of the input, most importantly the constraints to tuples expressed in the "where clause" of the query and how tuples are ranked.

The last years have brought up various research results [TCP09, SPGW10, ZEPS13, TCP14, TZES17] on reverse engineering database queries. Compared to existing approaches that operate on input in form of full tables, reverse engi-

Name	City	State	Plan	Month	Minutes	SMS	Data
John Smith	SF	CA	XL	June	654	87	1230
John Smith	SF	CA	XL	July	175	22	900
...	...	...	...	...	...	...	...
Jane O'Neal	LA	CA	XL	April	699	15	2300
Jane O'Neal	LA	CA	XL	June	334	10	1900
...	...	...	...	...	...	...	...
Richard Fox	Oakland	CA	XL	June	596	23	1272
...	...	...	...	...	...	...	...
Jack Stiles	San Jose	CA	XL	March	429	42	1192
Jack Stiles	San Jose	CA	XL	April	586	8	1275
...	...	...	...	...	...	...	...
Lara Ellis	San Diego	CA	XL	May	784	11	2107

Table 4.1: Sample relation of telecommunications traffic data

Lara Ellis	784
Jane O'Neal	699
John Smith	654
Richard Fox	596
Jack Stiles	586

Table 4.2: Example input list

neering top-k queries adds two complex ingredients to the re-engineering task. First, it is the rather small input, consisting of only a few (as k is usually quite short) ranked tuples and, second, the various ways top-k SQL queries can be formulated, given various sorting orders and aggregation functions.

Consider a relation Traffic, illustrated in Table 4.1, containing cellphone-traffic data. The relation contains textual attributes like name of the customer, the city and state the customer lives in, and the tariff plan and the month for which the traffic was realized. In addition, there are numerical attributes that measure the customer's traffic, like number of minutes talked, the number of text messages (SMS) sent, and the number of spent megabytes of data. Table 4.2 shows a top-k list with two columns and five rows. The input list does not have attribute names (or if it does, are not correlated to the attribute names in the database table). The first attribute is the customer's name, while the second is the performance attribute according to which the customer ranking was produced. Note that there are no empty cells in the list, all values are specified. Considering the Traffic relation of Table 4.1, we can see that the input ranking list can perhaps be generated using the following query:

```
SELECT name, max(minutes)
FROM traffic
WHERE state = 'CA'
GROUP BY name
ORDER BY max(minutes) DESC
LIMIT 5
```

This query computes the top-5 customers of the telecommunications company, living in the state of California, ranked by the number of minutes talked in a single month. In general, there can be several different queries that produce the same results; consider for instance augmenting the above query Q with an additional constraint to customers with the tariff plan 'XL', it would leave the result unchanged (including the order among tuples).

4.1.1 Problem Statement

Given a database D with a single relation R with schema $R = \{A_1, A_2, \ldots\}$ and an input relation L that represents a ranked list of items with their values. **The task we consider in this work is to efficiently and effectively determine queries Q_i that output tuples that resemble L when executed over R.**

We focus on top-k select-project queries over relation R of the form shown in Figure 4.1(a). We specifically focus on a single relation to emphasize on the intrinsic characteristics of top-k queries, instead of considering the reverse engineering of joins, too, which we address in Chapter 5.

```
SELECT id, agg(value)
FROM table
WHERE P1 and P2 and ...
GROUP BY id
ORDER BY agg(value)
LIMIT k
```

(a)

L

$L.e$	$L.v$
e	100
f	90
g	80
m	70
o	60

(b)

Figure 4.1: Query template and example input list L

The problem has two properties that can be relaxed or tightened. First, it can either demand determining only one, multiple, or all input-generating queries. Second, the notion of a query being valid in the sense that it resembles the input can be relaxed to a notion of approximately resembling the input.

The problem is challenging for the following reasons: (i) The size of the input list is rather small, it is difficult to derive meaningful (statistical) properties in order to identify valid predicates and ranking criteria, (ii) the relevant subset of R that features all tuples of the entities in L can become very large, and (iii) false positive and false negative candidate queries deteriorate system performance due to many necessary query evaluations and limit the chance to successfully determine a valid query that generates the input.

The presented approach, coined PALEO, is not limited to finding exact matches, but can also be adjusted and applied to finding queries that compute a ranking L' over R, with L' being similar to L. We get back to this generalization in Section 4.6. We refer to the specific attribute in R that contains the entities the table reports on as A_e and assume it is known a priori.

R	Base table in the database
A_i	Attribute in R
A_e	Entity attribute in R
L	Top-k input list
$L.e$	Entity column in L
$L.v$	Ranking column in L
e_i	Entities in A_e or $L.e$
v	Values in A_i
P	Predicate (atomic or conjunctive)
Q	Query
$Q(R)$	Result set of Q when querying R
$F(\tau_1, \tau_2)$	Footrule distance between two lists τ_1 and τ_2

Table 4.3: Overview of notations

As already indicated in the template query, we focus on predicates P of the form $P_1 \wedge P_2 \cdots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_i = v$ (e.g., $\mathsf{state} = \text{'CA'}$). Furthermore, we denote with *size* of a predicate $|P|$ the number of atomic predicates P_i in the conjunctive clause.

The input top-k list L has two columns; $L.e$ and $L.v$ denote the entity column and the numeric score column, respectively. Note that L does not contain the name of the column $L.v$ or the column name of $L.v$ is named for human consumption (e.g., "Total traffic", which can be total number of minutes, SMS, or data), i.e., not corresponding to the ones present in the database. Hence, referencing to the appropriate attribute in R cannot be done by name. Table 4.3 shows a summary of the most important notations used throughout this chapter.

4.1.2 Sketch of the Approach

A naive approach would enumerate all possible queries, say with a limited complexity of the predicate in the where clause, evaluate the queries one-by-one against the database and check whether the returned results resemble the input list. This is clearly beyond hope, even for relatively small databases and schemas.

Our approach, conceptually, loads all tuples from R that contain any of the entities in L. This table is called R' and is used in two subsequent steps, first, to determine the query predicate and, second, to find the right attribute(s) and aggregation function. In case R' is completely given, our approach is extremely effective in determining the individual building blocks of the desired query. When working on a subset of R', we show how to handle large amounts of potential candidate queries by introducing a suitability-driven order among them, in order to find the desired query early.

4.1.3 Contributions and Outline

With this work we make the following contributions:

- To the best of our knowledge, this work is the first to consider the problem of reverse engineering top-k OLAP queries. We present an efficient and effective solution to it, in a flexible and extensible framework.
- We show how to efficiently compute promising predicates using an apriori-style algorithm over R' and how to augment them with ranking criteria using data samples and statistics obtained from the base relation R.
- We present a probabilistic reasoning that allows ordering candidate queries by the likelihood that they compute the input ranking L. This, together with a method to skip unpromising queries dynamically at validation time, allows finding the desired valid queries very efficiently.
- We extend the methods to allow approximate matches within a distance measure threshold and present techniques for early elimination of many of the candidate predicates by using the properties of the distance measures.
- We report on the results of a carefully conducted experimental evaluation using data and queries from the TPC-H [TPC] and SSB [PO] benchmarks.

The rest of this chapter is organized as follows. Section 4.2 presents the framework and key ideas behind our approach, followed by the specific sub-problems of identifying query predicates in Section 4.3, and determining the ranking attributes and aggregation function, in Section 4.4. Section 4.5 considers handling changed data in R, and proposes a probabilistic model to rank queries by their expected suitability to generate the input. Section 4.6 describes extending the framework into allowing approximate matches and presents how this affects the specific sub-problems therein. Section 4.7 reports on the results of the experimental evaluation and presents lessons learned. Section 4.8 describes use-case application scenarios for the framework. Section 4.9 summarizes the work in this chapter.

4.2 Approach

The task of reverse engineering top-k queries is split into the following three steps, illustrated in Figure 4.2:

- **Step 1:** find the predicate P in the where clause of Q
- **Step 2:** find the ranking criteria
- **Step 3:** validate queries

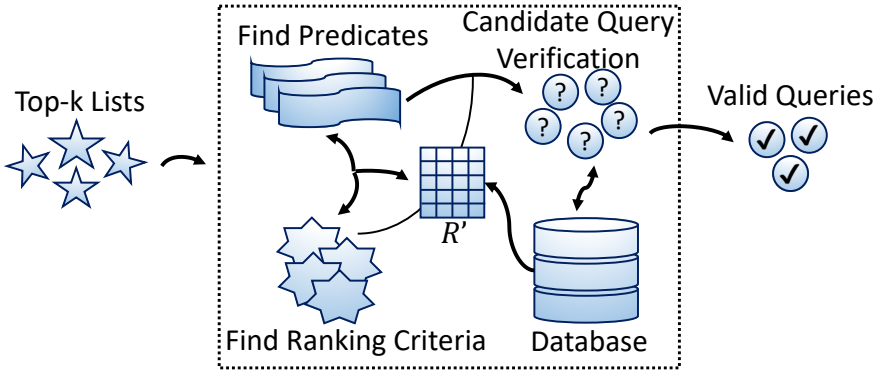


Figure 4.2: PALEO framework

As the basis of further computation, we first retrieve from relation R all tuples whose entity column contains one of the entities of the input table L; we call the resulting table R'.

4.2.1 Table R'

Consider a top-k list L as shown in Figure 4.1(b). Let $e_i \in \{e, f, g, m, o\}$ denote the entities in the column $L.e$.

By using a standard database index, such as a B+ tree, on the entity attribute of R, we can efficiently retrieve R' (shown in Table 4.4) containing all tuples from R matching any of the entities $e_i \in L.e$. Whether the index is actually used or the query optimizer decides to perform a table scan is not a concern here. In any case, in this example, the query to compute R' is:

SELECT * FROM R
WHERE A_e IN [e, f, g, m, o]

For the purpose of efficient access of its data, PALEO stores R' in-memory in a **column oriented** fashion, with columns being represented as arrays, allowing fast evaluation of aggregate queries over R'. The relation R' has $k' \geq k$ number of tuples, since it contains all tuples without (potentially) being filtered by predicates. In fact, it is reasonable to assume, without prior knowledge, that $k' \gg k$, as each distinct entity e_i can appear many times in R. We will allow to work on a subset (samples) of R' in Section 4.5, and study the consequences, but for now we assume R' in fact covers *all* tuples of *any* entity of the input.

4.2.2 The Three Steps

Candidate Predicates Identification. Using the tuples in R' we create a set of *candidate predicates* that are subsequently augmented with ranking criteria to make up full-fledged candidate queries.

Definition 4 Candidate Predicate
We say a predicate P is a candidate predicate iff for each entity that appears in L there is a tuple t in R' that fullfils the predicate. Formally,

$$\forall e_i \in L.e \, \exists \, tuple \, t \in R' : \, P(t) = true \, \wedge t.e = e_i$$

It is easy to see that a candidate predicate can potentially produce the top-k input list. In other words, having a candidate predicate in the where clause is a *necessary criterion* for a query to be a valid query, but it is *not a sufficient criterion*. This is because a candidate predicate can still "let through" tuples of other entities (that are not in the input table L) that can be ranked higher than the tuples in L, hence, the query is not a valid query as the output does not match the input.

Corollary 4.2.1 Downward-closure (anti-monotone) property of the candidate predicate criterion. *Given a predicate P_1 that is not a candidate predicate, then a predicate P_i such that $P_1 \subseteq P_i$ (that is, all sub-predicates in P_1 are also present in P_i) can not be a candidate predicate.*

The corollary follows immediately from the definition of candidate predicates: any predicate P_i with $P_1 \subseteq P_i$ for another predicate P_1 evaluates to true for a subset of tuples for which P_1 evaluates to true. This property is used to prune the search space in Section 4.3, similar to what the apriori algorithm [AS94] does for the support measure.

Ranking Criteria Identification. In the *second step* of our approach, we identify the ranking criteria according to which the entities in the top-k list are ranked. For this purpose we need to find a suitable numeric attribute (or multiple ones) including an aggregation function—or decide if one is used at all.

Definition 5 Candidate Ranking Criterion
*We say a ranking criterion, consisting of one or multiple numerical attributes and, if existing, an aggregation function is a candidate iff, **when executed on R'** together with a candidate predicate, it returns a **result identical to the input list L**.*

This definition is very reasonable, but similar to the criterion to identify candidate predicates, it is only a necessary condition to a valid ranking criteria for a query when executed over the entire relation R. It is, however, not a sufficient condition, as when executed on R there can be still other entities, not in L, that are disturbing the "correct" order. The case of partial matches is discussed in Section 4.6.

R'

$t.id$	E	A	B	C	$\cdots$	N_1	N_2	N_3	$\cdots$
1	e	a_1	b_9	c_3	$\cdots$	75	4	5	$\cdots$
2	e	a_1	b_8	c_1	$\cdots$	100	8	7	$\cdots$
3	e	a_3	b_1	c_6	$\cdots$	45	15	1	$\cdots$
4	f	a_1	b_8	c_1	$\cdots$	90	16	2	$\cdots$
5	f	a_5	b_4	c_6	$\cdots$	35	23	3	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
10	g	a_1	b_8	c_3	$\cdots$	80	42	14	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
20	m	a_1	b_8	c_4	$\cdots$	70	29	10	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
30	o	a_1	b_8	c_4	$\cdots$	60	31	7	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Table 4.4: Example of a relation R' for input L in Figure 4.1

Candidate Queries Identification and Evaluation. Using the candidate predicates and the valid ranking criteria we can form candidate queries. Each candidate query is executed on R and the results are compared with the input top-k list. The queries that produce instance-equivalent results with the original query are the valid queries.

4.3 Candidate Predicates

The task we consider in this section is to find all k-sized candidate predicates P_i. Each predicate can be simple atomic equality predicate like $(A = a_1)$ or conjunctions of atomic equality predicates, e.g., $(A = a_1) \wedge (B = b_8)$. Candidate predicates are determined over the table R', as described above. From Definition 4 we know that in order to be a candidate predicate, a predicate P has to have for *each* entity in the input L.*e at least one* tuple in R' with $P(t) = true$.

This criterion is anti-monotone (aka. downward-closed), i.e., a predicate P_i with size k can be considered a candidate predicate if and only if all its sub-predicates are also candidate predicates. This problem is similar to frequent itemset mining for which the apriori principle and algorithm [AS94] is widely known. In data mining terminology, itemsets resemble the values that are used to form the candidate predicates.

The method to compute candidate predicates in PALEO is described in Algorithm 4. In the first step, $k = 1$, we start by identifying all atomic candidate predicates, i.e., the predicates with size $|P_i| = 1$ (Lines 2–6 in Algorithm 4). For this purpose, for each column A_i we identify values v such that the predicate $P_i := (A_i = v)$ is a candidate predicate (Lines 3–4 in Algorithm 4). Furthermore, for each such created P_i we keep a set $\mathcal{I}_{P_i}$ containing the tuple ids (aka. row ids) that this predicate selects, i.e., $\mathcal{I}_{P_i} = \{t.id | P_i(t) = true\}$. In each additional step, conjunctive predicates of size k are created, by adding atomic

method: **findPredicates**
input: top-k list L
relation R'
output: a set of candidate predicates $\mathcal{P}$
1 $\mathcal{P} = \emptyset$; $k = 1$; $\mathcal{P}_k = \emptyset$
2 **for each** A_i **in** R'
3 **find** $P_i := (A_i = v)$ with $|P_i| = 1$ s.t.
4 $\forall e_i \in L.e \, \exists \, tuple \, t \in R' : \, P_i(t) = true \, \wedge t.e = e_i$
5 add P_i to $\mathcal{P}_k$
6 for each P_i keep $\mathcal{I}_{P_i} = \{t.id | P_i(t) = true\}$
7 **repeat**
8 $k = k + 1$
9 $\mathcal{P}_k = \emptyset$
10 **for each** $P_i \in \mathcal{P}_1$ and $P_j \in \mathcal{P}_{k-1}$ **and** $P_i \cap P_j = \emptyset$
11 **create** $\mathcal{I}_{P_{ij}} = \mathcal{I}_{P_i} \cap \mathcal{I}_{P_j}$
12 **if** $\mathcal{I}_{P_{ij}}$ covers all $e_i \in L.e$
13 add $P_{ij} := P_i \wedge P_j$ to $\mathcal{P}_k$
14 **until** $\mathcal{P}_k = \emptyset$
15 **return** $\mathcal{P} = \bigcup_k \mathcal{P}_k$

Algorithm 4: Finding candidate predicates

predicates from the set $\mathcal{P}_1$ to the predicates created in the previous iteration (Lines 7–14 in Algorithm 4). The algorithm does not create a predicate multiple times. The conjunctive predicate P_{ij} whose tuple ids set $\mathcal{I}_{P_{ij}}$ covers all entities in the input list is added to the set of candidate predicates with size k (Lines 12–13 in Algorithm 4) and will be used in creating candidate predicates of size $k + 1$ in the next iteration.

Example: Considering Table 4.4 and the input list in Figure 4.1, we create atomic predicates starting with column A as we iterate over its values a_i. Note that the entities in column E are sorted. The set of atomic candidate predicates is created, $\mathcal{P}_1 = \{P_1 := (A = a_1), P_4 := (B = b_8)\}$. These two predicates are candidates, since the tuples that fulfill the predicates cover all entities in the input list L. Furthermore, the set tuple ids that the predicates select are kept, e.g., $\mathcal{I}_{P_1} = \{1, 2, 4, 10, 20, 30\}$. If added as a selection condition, these candidate atomic predicates would result in a candidate query.

In each next step, we try to produce conjunctive clauses of size k from the predicates in $\mathcal{P}_1$ and $\mathcal{P}_{k-1}$. Thus, for $k = 2$, we test if the predicate $P_{14} := (A = a_1) \wedge (B = b_8)$ qualifies as a candidate by intersecting the corresponding sets of tuple ids. Since the intersected tuple ids in $\mathcal{I}_{P_1} \cap \mathcal{I}_{P_4} = \{2, 4, 10, 20, 30\}$ cover all entities in $L.e$, the predicate P_{14} is a candidate predicate. Recall that R' is held in memory and that we can, via tuple ids, very efficiently access the full tuple to check whether or not it matches the predicate.

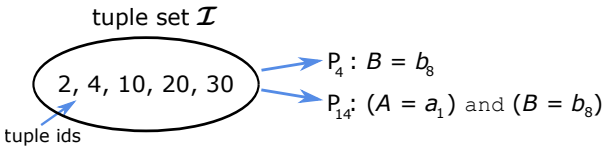


Figure 4.3: Mapping from tuple set to predicates

Properties of the algorithm:

(i) The algorithm is **correct** with respect to R', that is, predicates returned by the algorithm are guaranteed to be candidate predicates, following Definition 4. Further, the algorithm is **complete**, that is, it finds all possible candidate predicates over R'.

(ii) When predicates are applied in R instead of R' they can also let tuples with entities that are not in L pass, which leads to **false positive** candidate queries.

The difference to the apriori algorithm that operates on the support measure is that apriori counts the frequency of *all* itemsets and then determines the ones above the specified threshold. In our algorithm, we eliminate a predicate as soon as we find that it does not cover a certain entity. The same happens in each additional pass, since apriori will generate all the pairs of frequent items and count their appearance. Thus, all pairs that contain a false positive singleton will also be false positives.

4.3.1 Tuple Sets and Predicates

Some of the created candidate predicates have identical tuple sets $\mathcal{I}_{P_i}$. These predicates select the same tuples in R' and share the same data characteristics regarding to R'. Thus, candidate predicates are grouped according their tuple sets, i.e., if $\mathcal{I}_{P_i} = \mathcal{I}_{P_j}$, then P_i and P_j would belong to the same group.

Figure 4.3 depicts a tuple set mapped to a group of candidate predicates created from the tuples in Table 4.4. The predicates P_4 and P_{14} cover the same tuples in Table 4.4. Thus, for these predicates, it is enough to examine the data characteristics of the tuples in the tuple set $\mathcal{I}$.

4.4 Ranking Criteria

In order to find the ranking criteria according to which the ranking in the top-k list is done, PALEO operates on the distinct tuple sets, determined in the algorithm above. If relation R, and hence also R', is identical to the database state when the input data was once generated, it is guaranteed that PALEO is able to determine the valid ranking criteria.

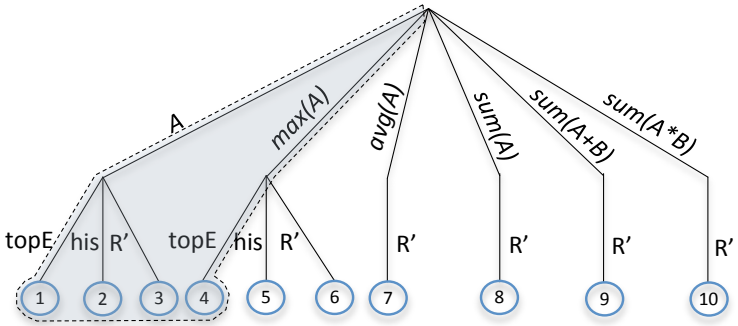


Figure 4.4: Order of looking for the ranking criteria

The actual size of R' depends naturally on the size k of the input list L and also on the data characteristics, i.e., how many tuples R contains for a single entity. We expect R' to be holding a factor of k/n less tuples than R, where n is the number of distinct entities in R, and that this allows to load R' entirely in main memory. While it might be reasonably cheap to execute a query on this R' in memory, note that we have to possibly do so very many times to identify suitable ranking criteria. That is, depending on the size of R' we can potentially reduce the runtime of our algorithm if it can be avoided to work on R' directly.

The idea is to harness small data samples, histograms, or simple descriptive statistics computed upfront from the base relation R in order to select a subset of potentially useful columns without touching R'. However, there might be invalid criteria identified or potentially also no criteria at all, given the limited coverage of data samples and the impreciseness of histograms. Therefore, identified candidate ranking criteria are validated on R' and in case no heuristic is applicable or was not successful, the whole ranking criteria identification is executed on R'.

Depending on the aggregation function we aim at checking for suitability, we can or cannot use some of these techniques. For instance, comparing the entities in L with the top entities stored for each column of R can be applied to queries with *max* aggregate function, but not directly to queries using *sum* as the aggregation function. Figure 4.4 summarizes this observation. Traversing the tree pre-order depth-first is the way PALEO looks for the ranking criteria, with the leaf nodes showing the order in which the techniques are applied. The system tries to identify the ranking criteria with smaller search space first. Thus, for instance, if the valid ranking criteria is $max(A)$ and comparing the top entities produces valid results, only the shaded part of Figure 4.4 will be processed.

4.4.1 Top Entities

The most apparent first attempt to identify an attribute according to which tuples are sorted in L is to store for each attribute in R the topmost entries, when sorted by the specific attribute. Then, we intersect the input entity set

from L with these top entries. More than just the k top values are stored to increase the chance that these entities do overlap with the entities in L. Clearly, it should also not be too large such that each numeric column appears promising. The exact way of how this idea is applied is shown in Algorithm 5, Line 6.

Before this is done, PALEO filters out attributes by applying three simple checks: it compares the max (min) values of the input list and the column and if the column's value is smaller (greater) than the max value of the input list it does not intersect the entities (Lines 3–4 in Algorithm 5). Additionally, the number of distinct values is compared; if the column has less distinct values than the input list, we skip this column (Line 5 in Algorithm 5).

The numerical columns that result in a *non-empty intersection* are considered as candidate numerical columns. Thus, using R' and the tuple sets created in finding candidate predicates, they are checked whether they can match the ranking in the top-k input list.

4.4.2 Querying Histograms

In the case no candidate numerical columns have been identified with the above intersection of top entities, PALEO employs histograms describing an attribute's frequency distribution in order to find candidate attributes that appear suitable for ranking. As we consider only numeric attributes to be used as the bases of ranking criteria, such a histogram describes how frequent a specific numeric value appears in the attribute's column in relation R. One idea is comparing the value-frequency distributions of the histogram of the input list with the histograms of the numerical columns in R, by using histograms distance measures such as Earth Mover's Distance [RTG98]. However, a top-k list is inherently small and does not contain enough elements to provide a meaningful distribution. Hence, PALEO samples each attribute's histogram and calculates the L1 distance between its top-k values and the input values. Similar to using top entities of each column, we draw samples following the distribution described in the histogram. PALEO uses equi-width histograms having 1000 cells each.

The process is shown in Algorithm 6. This is done for each attribute, which allows ordering all attributes by the L1 distance of the sampled data to the data in L. Depending on the data in the table, if there is a column with similar values and distribution as the column we are looking for, it is possible that the correct column does not have lowest L1 distance. In order to account for this, we consider the top 30% of the columns in the list as candidate attributes.

4.4.3 Validation over R'

As a validation for the possible ranking criteria identified above, we use the tuples in R'. In the case when we have successfully identified candidate attributes with the previous techniques, we first check if any of these candidate attributes can produce the ranking. For this purpose, we go through the distinct tuple

method: **topEntities**
input: top-k list L
relation R'
output: a set of candidate numerical columns $\mathcal{A}_C$
1 **for each** A_i **in** R'
2 **if** A_i not numerical, **then** skip A_i
3 **if** $\max(v \in A_i) < \max(v \in L.v)$, **then** skip A_i
4 **if** $\min(v \in A_i) > \min(v \in L.v)$, **then** skip A_i
5 **if** $|A_i| < |L.v|$, **then** skip A_i
6 **if** $TopE(A_i) \cap L.v \neq \emptyset$, add A_i to $\mathcal{A}_C$
7 **return** set of candidates $\mathcal{A}_C$

Algorithm 5: Finding candidate columns with top entities

sets $\mathcal{I}_i$ computed in Section 4.3 and check which of the candidate numerical columns, i.e., their sorted aggregated values exactly match the input L.

Some of the supported ranking criteria cannot be identified by the above mentioned techniques, requiring more complicated statistics and this is beyond the scope of this work. For instance, with the *avg* and *sum* aggregate functions, the top entities for a column depend heavily on the predicate, since the values are aggregated over multiple tuples. Similarly, harnessing histograms with *sum* would involve convolutions of the histograms of the pairs of columns.

As a fall back, if none of the candidate attributes can produce the ranking criteria, we revert to checking the remaining numerical columns in R' that were not found as candidates. We still use only the tuples with tuple ids found in the tuple sets $\mathcal{I}_i$. For each tuple set and each numerical attribute in R' that passes our (three) simple checks (i.e., min, max comparison, and number of distinct items), we compute whether the tuples in $\mathcal{I}_i$ if sorted according to the specific attribute and aggregate function are identical to L. After identifying the appropriate numerical attribute and aggregate function, we can filter out some of the *candidate predicates*. If a certain tuple set does not contain the input numerical values, we **remove** this tuple set and all the candidate predicates that correspond to it from the candidate predicates.

4.5 Handling Variations of R

The techniques behind PALEO discussed so far are based on the assumption that exactly the same relation R that produced the input list L is available and that it is feasible to operate on it directly. However, it might appear that tuples in R have changed, for instance, because of inserts, updates, and deletes, due to slowly changing dimensions [Kim96] in data warehousing scenarios, or only a subset (sample) is available. In this section, we describe how PALEO deals with situations when only subset of the original tuples in R is available.

method: **sampleHistograms**
input: top-k list $L(L_e, L_v)$
relation R'
row sets I_i
output: a set of candidate numerical columns A_C

```
1   sampleSize = α × k
2     for each A_i in R
3       if A_i not numerical, then skip A_i
4       if max(v ∈ A_i) < max(v ∈ L_v), then skip A_i
5       if min(v ∈ A_i) > min(v ∈ L_v), then skip A_i
6       if |A_i| < |L_v|, then skip A_i
7       get a sample from histogram(A_i)
8       compute d(TopK(sample), L_v)
9       put [d, A_i] in A_C
10      sort A_C in asc order of d
11  return top 30% of A_C
```

Algorithm 6: Finding candidate columns with sampling histograms

This assumption has direct consequences on PALEO's ability to accurately identify suitable predicates and ranking criteria. As we have discussed above, determining query predicates with the proper table R at hand *only* leads to obtaining **false positives** in the candidate predicates, introduced by additional entities outside R' that qualify for the predicate. The changed data further introduces **false negatives**. That is, the query that generated the input might not be found at all, although such a query exists. This is caused by missing or modified tuples in R that would be required to unveil a predicate to be fulfilled by all of the k entities. False negatives are synonym to *loss in recall*, i.e., the fraction of found queries to all existent queries that generate the input.

We address this by:

- Reasoning about likelihood of being a successful query.
- Smart evaluation to skip unpromising queries.

Variations in R means also variations in R'. Let us denote the table stemming from the modified base table as R''. It can happen that R'' does not contain tuples from all entities from the input list, for instance if all tuples for a certain entity $e_i \in L.e$ have been deleted from R. Recall that the method for finding predicates, described in Algorithm 4 demands that a predicate must cover all entities of the input list $L.e$.

Now, it is possible that the tuples containing the valid predicate for a certain entity have changed in the columns that comprise the predicate. Then, it is impossible to precisely validate or invalidate the predicate using the method

in Algorithm 4: being strict, missing the tuples with the valid predicate for a certain entity will lead to evicting the valid predicate even though the majority of entities in R'' contain tuples with it, thus resulting in false negatives. To avoid that, the condition of evicting a predicate is relaxed. Instead of demanding that a predicate is considered as a candidate predicate if it covers all distinct entities in R'', we ask for it to cover the *majority of the entities*, thus taking into account that some entities can have tuples with the valid predicate missing. Another possible approach is not to evict predicates at all, i.e., form all the predicates that we encounter in R'' while not demanding any entity covering. This might, however, result in very many candidate predicates with too many false positives. Executing candidate queries for all such predicates will drastically decrease the overall efficiency of PALEO.

We describe a probabilistic model of assessing candidate predicates when the data in the base table has changed and how uncertainty in finding ranking criteria can be handled.

4.5.1 Assessing Candidate Predicates

Changes in R introduce uncertainty in finding the valid predicates. To account for such changes, the condition of evicting a predicate is relaxed. As a result, our methods identify more candidate predicates that need to be assessed whether or not they are likely to be indeed a valid predicate. This assessment is later used when executing queries in the final step such that queries can be executed in increasing order of the likelihood to be in fact a valid query.

A candidate predicate P_i identified from the table R'' is a false positive if: $\exists\ e_i \nexists\ t$ s.t. $P(t) = true$. In other words, if for a certain entity e_i there is no tuple for which the predicate P is valid, then this predicate is a false positive. This means that a query with this predicate would return a top-k list without the entity e_i.

Consider a predicate P over the attributes $A_1, \ldots A_m$. The probability that a tuple exists in relation R is given by the number of distinct entries of the columns A_i (i.e., $|A_i|$) as

$$P[\text{tuple exists}] = \prod_i \frac{1}{|A_i|}$$

Consider an entity e_j for which we did not find a tuple that matches the predicate and let $unseen(e_j)$ be the number of changed tuples of entity e_j, then

$$P[\text{won't see for } e_j] = (1 - P[\text{tuple exists}])^{unseen(e_j)}$$

The probability that at least one entity is rendering this predicate to be a false positive (by not providing a matching tuple) is thus given as

$$P[\text{false positive}] = 1 - \prod_j (1 - P[\text{won't see for } e_j])$$

4.5.2 Approximating Ranking Criteria

Operating on R'' also introduces uncertainty in finding ranking criteria. Since not all tuples for each entity e_i are the same, the ranking criterion cannot exactly match the numerical values in the input top-k list. This is why there is a need of measuring the suitability of each candidate ranking criteria to the input list. For this purpose, we compute the distance between the input values and the candidate attribute(s) values. We use the L1 distance (aka. Manhattan distance) that is simply the sum of absolute differences in the numeric values.

Queries without Sum. The changes in the tuples for an entity e_i renders the **topEntities** method (Section 4.4.1) not directly applicable. Without the identical tuples, it is difficult to match the candidate numerical columns with the input ranking values. Using the L1 distance and the column values in R'' (Section 4.4.3) provides the possibility to compute the suitability of the candidate ranking columns. That way, each candidate column has a corresponding L1 distance that is used in ranking the candidate queries.

Queries with Sum. The *sum* aggregate function sums up all values for a certain entity e_i. Since with changed data some of the tuples for an entity are missing, they need to be approximated. We do this by using the column values for the column(s) in R''. Using this approximation, the L1 distance to the input ranking values is calculated and then used for ranking the suitability of the column(s).

The approximation of the sum for each entity is done using the tuple id sets. We take a look at the more complicated case of having a sum of two columns A_i and A_j. Thus, for a predicate P with a corresponding tuple set $\mathcal{I}_P$, for each entity e_i let $sum_{A_{ij}}(\mathcal{I}_P)$ denote the sum of the values, of the columns A_i and A_j, of the **tuples in** R'' with tuple ids in $\mathcal{I}_P$ that have an entity e_i, i.e.:

$$sum_{A_{ij}}(\mathcal{I}_P) = A_i(\mathcal{I}_P)\ op\ A_j(\mathcal{I}_P)\ \ s.t.\ t.e = e_i\ ,\ \ op \in \{+, *\}$$

Additionally, let $\#v$ denote the number of tuple ids in the tuple set $\mathcal{I}_P$ of the entity e_i, i.e., the number of tuples that the predicate P selects with e_i. We approximate the sum as:

$$appxSum_{e_i}(\mathcal{I}_P) = \frac{sum_{A_{ij}}(\mathcal{I}_P)}{\#v} \times (\#v \times \frac{|e_i|_R}{|e_i|_R - unseen(e_i)})$$

where $|e_i|_R$ is the number of tuples in R for the entity e_i. Thus, for each entity e_i, the average summed value from the unchanged tuples is multiplied with the approximated selectivity of the predicate P. The sorted list $appxSum(\mathcal{I})$ formed from the sums for each entity $appxSum_{e_i}(\mathcal{I}_P)$ is then used for calculating the L1 distance d to the input list and ranking the candidate column pairs.

4.5.3 Combined Model

The queries formed from the combination of candidate predicates and ranking criteria need to be validated by executing them on R. The order of execution is done ordered by a suitability value for each candidate query Q_c. The suitability is computed as:

$$s(Q_c) = (1 - P[\text{false positive}]) \times (1 - d)$$

where P[false positive] is the probability of the predicate in the candidate query of being a false positive and d is the max normalized L1 distance between the ranking criteria in Q_c and the numerical values in the input list L.

4.5.4 Working with Samples of R'

Consider a scenario where it is impossible or unfeasible to work on the complete relation R' (the subset of R of all tuples that contain any of the entities in L). This relation R' can be very large, potentially as big as R, if there are many tuples for each distinct entity—a typical case in data-warehousing applications that often aggregate large amounts of observations of a specific entity. The probabilistic model for assessing candidate predicates together with the approximation of the ranking criteria can also be applied to such a scenario as well.

We consider two approaches of sampling. First, we sample by retrieving all tuples for a certain (e.g., randomly selected) subset of the entities in L.*e*. In this way, we do not get any false negatives and the candidate predicates set is a superset of the valid predicates. This is because having all tuples in R'' for a certain entity is guaranteed to contain the tuples with valid predicates. As a result, our algorithm will create the predicate as a candidate. However, the drawback of this approach is having too many false positives. This can especially happen if for a sampled entity there are too many tuples in the base table R. This will lead to creating a large amount of false positives which impairs efficiency.

Sampling uniformly from all entities mediates this problem, thus sampling a certain percentage of the tuples from each entity. This way, possibility of false positives is decreased, at the price of an increased possibility of false negatives. We encounter the same problem as if we would sample by tuple: it can happen that tuples that contain a valid predicate are not sampled for a certain entity. Relaxing the condition of evicting a predicate mediates this problem.

We can draw a parallel between the scenarios of having modified data in R and sampling. The tuples that are sampled in R'' correspond to the tuples in the base table that have the columns comprising the valid predicate **unmodified**. Hence, the not sampled tuples are analogous to the ones that are modified.

Consider a predicate P_i that is a valid predicate for an entity e_i. The probability that k tuples with the valid predicate are sampled in R'' has a hyperge-

ometric distribution, i.e.,

$$P[\text{k tuples sampled}] = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{K}}$$

where K is the total number of tuples with the predicate in R', N is the total number of tuples to sample from, i.e., $N = |R'|$, and $n = |R''|$ is the number of sampled tuples.

The probability of sampling at least one tuple with the valid predicate P_i for an entity e_i:

$$P[\text{one tuple sampled}] = 1 - \frac{\binom{K}{0}\binom{N-K}{n-0}}{\binom{N}{K}}$$

Considering an input top-k list with m distinct entities e_i and assuming independence in the sampling from the different entities, the probability of seeing a tuple with the valid predicate is:

$$P[\text{all } e_i] = P[\text{one tuple sampled}]^m$$

Intuitively, this probability describes that increasing the sampling size increases the probability of sampling a tuple with the valid predicate for each distinct entity. Consequently, making the condition of evicting a predicate more strict as the sample size increases is needed, i.e., increasing the number of entities e_i that are covered by a predicate so it can qualify as a candidate predicate. This would eliminate the creation of too many false positives with larger sample sizes.

4.5.5 Smart Query Validation

Ordering candidate queries by their expected suitability to answer the input L promises to find a valid query early—ideally at the first query execution. Even if more than one valid query is to be found, such an order is accelerating the discovery process immensely. We will show in the experimental evaluation that this is indeed the fact.

Now, instead of purely trusting the order, it would be careless to simply execute queries sequentially in the given order, without trying to benefit from information learned while executing them. Consider a candidate query Q_c that is executed and yields a result $Q_c(R)$ that is very similar to the input list L, but is still not an exact match. It would be preferable to continue validating queries that are similar to Q_c and skip those in the ordered query candidate list $\mathcal{C}$ that are not.

It is clear that the similarity (overlap) of the results of a candidate query when executed over R and input list L can be directly computed, using Jaccard similarity for instance. But for the not-yet-executed queries we do not have direct insight on their result, but we can "speculate" about it: we model this

method: **resultDrivenValidation**
input: ordered list of candidate queries $\mathcal{C}$;
Jaccard similarity threshold τ
output: a valid query Q_v

```
1   Q_c := C.first
2   /* search for first query with results overlapping L*/
3   while J(Q_c(R).e, L.e) < τ
4       Q_c := C.next
5   /* keep this first match query */
6   Q_fm := Q_c
7   foundR := false
8   foundR := true if J(Q_c.v, L.v) > τ
9   while(C.hasNext)
10      Q_c = C.next
11      /* skip query Q_c? */
12      if (P(Q_c) ∩ P(Q_fm) = ∅ or
                (foundR and R(Q_fm)! = R(Q_c))))
13          continue
14      execute Q_c
15      return if found valid
16  resultDrivenValidation(skipped Q_c)
```

Algorithm 7: Result driven candidate query validation

similarity between Q_1 and Q_2 by two means; first, by the common atomic equality predicates in the conjunctive where clause, and second, by the use of the same (or not) ranking criteria. For this, with $R(Q)$ we denote the ranking criterion of a query and with $P(Q)$ the set of its atomic predicates.

For each executed query we check if its output matches the input list. In the first part of the algorithm presented in Algorithm 7, we sequentially test the candidate queries until we have found for which the entities in its results are similar to the entities in $L.e$. This query is denoted Q_{fm}, for "first match query". We also check if the numeric values of the query result are similar to the numeric values $L.v$ of the input list L, again, using the Jaccard similarity. If they are sufficiently similar, we mark the ranking criteria of query Q_{fm} as valid. In the second while loop (Lines 9–13 in Algorithm 7), we iterate over the remaining candidate queries and skip those queries whose predicates are not at all overlapping with the predicates in Q_{fm}. We further skip queries that have a different ranking criterion to the one of Q_{fm} (Line 12 in Algorithm 7), in case this was found as valid.

If by the end of the query list $\mathcal{C}$ a valid query is not found, the algorithm is called for the previously skipped queries, until all queries are evaluated or one valid query is found.

4.6 Computing Similar Entity Rankings

Our approach to reverse engineering queries can be relaxed to allow finding also approximately matching queries, i.e., queries that produce result lists that are similar to the input L. This can be useful in a data exploration scenario, where users can input a list without knowing specific details about the entities in the dataset, or in cases where L has been generated in the extreme, through crowdsourcing top-k rankings. Thus, in looking for approximate matches, we consider a single-column input list L containing only an ordered list of entities (i.e., the column $L.e$ in Figure 4.1). The numeric score of the entities is not known. The task is to efficiently and effectively determine queries Q that, when executed over the database, compute result lists that are **similar** to L. A user-defined **similarity threshold** θ is used to control how similar the found queries, respectively their results, should be, relative to L.

This task introduces several new challenges that need to be addressed. First, we need to find predicates and ranking criteria, when the score (ranking value) of the entities is **not known**. Thus, the set of candidate predicates that (partially) satisfy the input list is expected to significantly grow, requiring novel effective techniques for finding and pruning unpromising predicates. Second, finding similar rankings requires the application of distance measures like Kendal's tau and Footrule distance [FKS03]. Instead of generating all possible queries, evaluating them, and checking if they comply to the similarity threshold, we show how to embed distance-measure specific bounds into the query generation process.

To compare the top-k lists we use Spearman's Footrule [FKS03], a prominent distance measure used for comparing top-k list, described in Section 2.3.2. Spearman's Footrule is the L1 distance between two top-k lists, i.e., for two top-k lists τ_1 and τ_2, Spearman's Footrule $F(\tau_1, \tau_2) = \sum_i |\tau_1(i) - \tau_2(i)|$ where $\tau_1(i)$ denotes the position of entity i in τ_1. For entities found in only one of the lists we take an artificial position of $k + 1$ as specified in [FKS03].

4.6.1 Approach

Allowing for approximate matches changes each of the three steps presented in Section 4.2: finding the predicates, the ranking criteria, and validating and ranking the candidate queries. However, all computations are still done on R' which is kept in-memory.

4.6.2 Candidate Predicates

First, the definition of candidate predicate needs to be adjusted to fit the modified task.

Definition 6 Candidate Predicate
We say a predicate P is a candidate predicate in computing similar entity rankings iff for at least one entity that appears in L there is a tuple t in R' that

fulfills the predicate. Formally,

$$\exists e_i \in L.e \; \exists \, tuple \, t \in R' : \; P(t) = true \; \wedge t.e = e_i$$

Note that this definition underpins a big difference to Definition 4 that would insist that each (not at least one) entity in $L.e$ has a tuple that fulfills the predicate.

Now, the table R' contains only tuples where the entity attribute $R'.e$ is one of the entities from $L.e$. Thus, Definition 6 implies that all distinct attribute:value pairs in R', like (state='MN') or (date='10.10.2015'), are atomic candidate predicates of size $|P| = 1$. Candidate predicates of size larger than $|P| = 1$ can be generated simply by forming the conjunction of the atomic predicates belonging to the same tuple, e.g., (state='MN') $\wedge$ (date='10.10.2015').

A naive approach would simply take all these candidate predicates and advance to the next step of finding ranking criteria. This would, however, result in a drastically reduced performance, inducted by a radical expansion of the set of candidate predicates. We can do much better by investigating whether or not a candidate predicate is at all capable or likely of being capable of leading to a query that is satisfying the similarity threshold. More precisely, to reduce the number of candidate predicates, we rely on the fact that the user expects queries resulting in lists **similar** to the input, i.e., those queries for which $F(Q(R), L) \leq \theta$, for reasonably small values of θ. This means that a candidate predicate that is fulfilled by only one entity in L is presumably not a good candidate since the resulting list, irrespective of the ranking criteria, would have only one overlapping entity with the input, and therefore will not satisfy the user's information need.

What we need to know in order to achieve this is how many overlapping entities μ, the rankings $Q(R)$ and L need to have such that $F(Q(R), L) \leq \theta$. This then allows us to eliminate candidate predicates whose tuples cover less than μ entities in $Q(R)$. According to [MAM15], two ranked list τ_1 and τ_2 with Footrule distance $F(\tau_1, \tau_2) \leq \theta$ must have an overlap:

$$\mu \geq \lceil 0.5 \times (1 + 2k - \sqrt{1 + 4\theta}) \rceil$$

Using the minimum number of overlapping entities μ, we can already eliminate many of the candidate predicates. However, as we show in the experiments in Section 4.7, still many useless predicates were generated. In order to eliminate even more candidate predicates, we further leverage the properties of the Footrule distance. We first define the concept of **displacement of an entity** $\boldsymbol{e}$, ω_e as:

Definition 7 Displacement of an Entity
For two top-k lists τ_1 and τ_2, $\tau_1 \neq \tau_2$, we define the displacement, denoted with ω_e, of an entity $e \in D_{\tau_1} \cap D_{\tau_2}$, as the difference of the position of the entity

in the two rankings, i.e., $\omega_e = |\tau_1(e) - \tau_2(e)|$*. In the case when* $e \in D_{\tau_1} \setminus D_{\tau_2}$*,* $\omega_e = k + 1 - \tau_1(e)$ *or vice versa.*

The Footrule distance of two lists τ_1 and τ_2 is in fact a sum over the displacements of the entities in $D_{\tau_1} \cup D_{\tau_2}$. Next, with the following theorem we define the maximum value for the displacement of any entity e in any two list τ_1 and τ_2, where $F(\tau_1, \tau_2) = \lambda$.

Theorem 4.6.1 *The maximum displacement of an entity* e *for two top-k lists* τ_1 *and* τ_2 *of size* k *with Footrule distance* $F(\tau_1, \tau_2) = \lambda$ *is:*

$$max(\omega_e) = min\{\lfloor \frac{\lambda \times k \times (k+1)}{2} \rfloor, k\}$$

Proof Restricting the maximum displacement $max(\omega_e)$ to k is clear. Considering the fact that the rankings are of size k (have k entities), by the definition of a displacement it follows that the $max(\omega_e)$ for two rankings τ_1 and τ_2 cannot be larger than k for any distance $F(\tau_1, \tau_2)$.
Now, let us look at the case of showing that the maximum displacement $max(\omega_i)$ for any two rankings τ_1 and τ_2 with distance $F(\tau_1, \tau_2) = \lambda$, cannot be larger than $\frac{\lambda \times k \times (k+1)}{2}$. Top-k rankings are bijections from the set D_{τ_1} to itself, where $D_{\tau_1} \neq D_{\tau_2}$ for two top-k lists τ_1 and τ_2. The Footrule distance, in this case, is a sum not only over the misplacement of the overlapping entities, but over the displacement of the missing entities as well. Since we are working with lists of same size, we have the same number m of missing entities in τ_2 and τ_1. Therefore, if we ignore the fact that the missing entities are different in the two top-k lists, and just consider that we have m missing entities in each list and $k - m$ common entities, there is a bijection between the set of k entities and the lists τ_1 and τ_2. Now, let us assume that $max(\omega_i)$ is larger than $\frac{\lambda \times k \times (k+1)}{2}$. If $max(\omega_e) > \frac{\lambda \times k \times (k+1)}{2}$ then we have to have at least one more entity j displaced in order to keep the bijection between the two sets. Since $F(\tau_1, \tau_2) = \sum_{e \in D} \omega_e$ it follows that we have at least one entity whose displacement is $\omega_j < \frac{\lambda \times k \times (k+1)}{2}$, or we have more than one displaced item, but the sum of their displacements is smaller than $\frac{\lambda \times k \times (k+1)}{2}$. However, this would break the bijection relation between the two permutations, which is not possible, proving that $max(\omega_e) \leq \frac{\lambda \times k \times (k+1)}{2}$.

Therefore, it holds that $max(\omega_e) = min\{\lfloor \frac{\lambda \times k \times (k+1)}{2} \rfloor, k\}$. ∎

Knowing the maximum displacement allows us to find a set of entities $e \in L$ that must be covered by candidate predicates. For instance, when we have an input top-k list L of size 10, and we want to find all queries resulting in lists $Q(R)$ such that $F(Q(R), L) \leq 0.1$, all the candidate predicates must cover the first 5 entities since the maximum displacement $max(\omega_e)$ is 5 and thus the maximum position of an entity that appears in L but not in $Q(R)$ is $k + 1 - max(\omega_e) = 6$.

method: **findPredicatesWithDistance**
input: top-k list L
relation R'
overlap μ
max displacement $max(\omega_e)$
output: a set of candidate predicates $\mathcal{P}$
1 $\mathcal{P} = \emptyset$; $n = 1$; $\mathcal{P}_n = \emptyset$
2 $\mathcal{M}_e$ =top-$(k - max(\omega_e))(L)$
3 **for each** A_i **in** R'
4 **find** $P_i := (A_i = v)$ with $|P_i| = 1$ s.t.
5 $\exists e_i \in L.e \, \exists \, tuple \, t \in R' : \; P_i(t) = true \; \wedge t.e = e_i$
6 **if** $count(e_i) \geq \mu$ **and** $\mathcal{M}_e \subseteq \{e_i\}$
7 add P_i to $\mathcal{P}_n$
8 for each P_i keep $\mathcal{I}_{P_i} = \{t.id | P_i(t) = true\}$
9 **repeat**
10 $n = n + 1$
11 $\mathcal{P}_n = \emptyset$
12 **for each** $P_i \in \mathcal{P}_1$ **and** $P_j \in \mathcal{P}_{n-1}$ **and** $P_i \cap P_j = \emptyset$
13 **create** $\mathcal{I}_{P_{ij}} = \mathcal{I}_{P_i} \cap \mathcal{I}_{P_j}$
14 **if** $count(e_i) \in \mathcal{I}_{P_{ij}} \geq \mu$
15 **and** $\mathcal{I}_{P_{ij}}$ covers all $e_i \in \mathcal{M}_e$
16 add $P_{ij} := P_i \wedge P_j$ to $\mathcal{P}_n$
17 **until** $\mathcal{P}_n = \emptyset$
18 **return** $\mathcal{P} = \bigcup_n \mathcal{P}_n$

Algorithm 8: Finding candidate predicates in computing similar entity rankings

Using these two optimizations, we propose a modified approach of the one shown in Algorithm 4 that generates a set of candidate predicates. This new approach is described in Algorithm 8.

The method requires as input the number of overlapping entities μ and the max displacement of an entity $max(\omega_e)$. These values depend only on the size of the input list k and the threshold θ, and can be calculated using the formulas presented above. The method starts by creating the set $\mathcal{M}_e$ which will contain the top-$(k - max(\omega_e))$ entities in the input list L (Line 2 in Algorithm 8). These are the entities that must be covered by each candidate predicate P_i. In the first step, the method creates all atomic candidate predicates, i.e., those with size $|P| = 1$ (Lines 3–8 in Algorithm 8). For each column A_i we identify values v such that the predicate $P_i := (A_i = v)$ will satisfy two conditions (Line 6 in Algorithm 8). First, it must cover at least μ number of entities, which ensures that the tuples in R' that fulfil this P_i contain the minimum number of overlapping entities. Second, the predicate must cover all of the mandatory entities

Best Buy
Cargill
Target
Valspar
Kroger

Table 4.5: Example input list L

R'

$t.id$	Name	State	Date	⋯	Sales USD	⋯
1	Best Buy	MN	10.10.2015	⋯	818109	⋯
2	Best Buy	MN	11.10.2015	⋯	51369	⋯
3	Best Buy	AZ	09.10.2015	⋯	42101	⋯
4	Best Buy	CA	10.10.2015	⋯	12256	⋯
5	Best Buy	CA	15.10.2015	⋯	46423	⋯
	⋯	⋯	⋯	⋯	⋯	⋯
10	Cargill	MN	12.10.2015	⋯	523087	⋯
11	Cargill	CA	10.10.2015	⋯	256102	⋯
	⋯	⋯	⋯	⋯	⋯	⋯
20	Kroger	MN	10.10.2015	⋯	101022	⋯
	⋯	⋯	⋯	⋯	⋯	⋯
30	Target	MN	12.10.2015	⋯	221003	⋯
	⋯	⋯	⋯	⋯	⋯	⋯

Table 4.6: Example of a relation R' for input L in Figure 4.5

in $\mathcal{M}_e$. In addition, for each such predicate that satisfies these two conditions, we keep a set $\mathcal{I}_{P_i}$ that contains the tuple ids that this predicate selects, i.e., $\mathcal{I}_{P_i} = \{t.id | P_i(t) = true\}$. In each additional step, the algorithm creates predicates of size n by creating conjunctions of the atomic predicates from the set $\mathcal{P}_1$ created in the previous iteration (Lines 9–17 in Algorithm 8). None of the predicates is created multiple times. Each of the conjunctive predicates P_{ij} whose tuple ids set $\mathcal{I}_{P_{ij}}$ covers at least μ entities and in addition covers all mandatory entities in $\mathcal{M}_e$, is added to the set $\mathcal{P}_n$ which contains the predicates with size n. This set will be used in the next iteration in creating candidate predicates with size $n + 1$.

Example: Let us consider Table 4.6 and the input list in Table 4.5. Furthermore, let $\mu = 3$ and the max displacement $max(\omega_e) = 3$. The method starts by initializing the set with mandatory entities $\mathcal{M}_e$ with the top-2 entities from the input list, i.e., 'Best Buy' and 'Cargill'. Then it starts iterating over the values in the columns, e.g., state, date, etc. Thus, it creates the set of atomic candidate predicates, $\mathcal{P}_1 = \{P_1 :=$(state='MN'), $P_2 :=$(date='10.10.2015')$\}$. Both of these predicates cover at least 3 entities and cover the mandatory entities in $\mathcal{M}_e$. In addition, the tuple ids sets that correspond to this predicates are kept, e.g., $\mathcal{I}_{P_1} = \{1, 2, 10, 20, 30\}$. In each additional step, conjunctive clauses of size n are produced by using the predicates in $\mathcal{P}_1$ and $\mathcal{P}_{n-1}$. Thus, for $n = 2$ the predicate $P_{12} :=$(state='MN')$\wedge$ (date='10.10.2015') is tested whether it qualifies

method: **findSum**
input: top-k list L
relation R'
tuple sets $\mathcal{I}_i$
threshold θ
output: a pair of numerical columns (A_i, A_j)
1 **for each** $(A_i, A_j) \in R'$
2 **for each** tuple set $\mathcal{I}_i$
3 **for each** distinct entity e_i
4 $A_{ij_{sum}}(\mathcal{I}_i) = A_i(\mathcal{I}_i) + A_j(\mathcal{I}_i)$
5 group by e_i
6 add $A_{ij_{sum}}(\mathcal{I}_i)$ to $Q(R')$
7 sort $Q(R')$
8 **if** $F(Q(R'), L) \leq \theta$
9 **return** the valid pair (A_i, A_j)

Algorithm 9: Finding $sum(A + B)$ ranking criterion

as a candidate by intersecting the corresponding tuple ids sets. The intersected tuple ids $\mathcal{I}_{P_1} \cap \mathcal{I}_{P_4} = \{1, 20\}$ do not cover at least 3 entities from the input list, thus the predicate P_{12} is not a candidate predicate. Since R' is kept in main memory, by using the tuple ids the method efficiently can access the full tuple to perform the checks needed.

4.6.3 Ranking Criteria

Once we have identified the set of candidate predicates, the next step is to identify the ranking criteria according to which the entities in the top-k lists are ordered. For this purpose, we need to find a suitable numeric attribute (or multiple ones) including an aggregation function—or decide if one is used at all.

Definition 8 Candidate Ranking Criterion
We say a ranking criterion, consisting of one or multiple numerical attributes and, if existing, an aggregation function is a candidate iff, ***when executed on*** $\boldsymbol{R'}$ *together with a candidate predicate, it returns a resulting list* $Q(R')$ *such that the resulting list is similar to the input list* L*. More formally, for a given similarity threshold* θ*, a ranking criterion is a candidate iff* $F(Q(R'), L) \leq \theta$ *.*

As we do not know the ranking values of the entities in the input list, practically all the combinations of numeric attributes and aggregate functions could qualify as candidates. Only after executing the query and comparing its result to the input list, we could classify the query as a candidate or dismiss it. Since executing the query is in fact the most expensive part of PALEO, we need to come up with techniques that would allow us an earlier elimination of the non-qualifying ranking criteria.

For the purpose of identifying the candidate ranking criteria we leverage the table R' which is held entirely in main memory. We use the tuple sets $\mathcal{I}_i$ identified in Section 4.6.2 and check which ranking criteria, i.e., numerical column(s) in combination with the supported aggregate functions, can produce a sorted list of entities similar to the input list L. This corresponds to executing queries on R', where each query is a combination of the candidate predicates created with Algorithm 8 and the ranking criteria supported by the system. The sets of tuple ids are used as sort of an index to directly access the tuples in R'. The top-k lists that result from these queries are compared with the input list L by calculating the Footrule distance. Each of the queries whose resulting list has a Footrule distance at most θ, is created as a candidate query Q_c that needs to be executed on the base table R. All other queries are discarded. In this way, we manage to prune many of the candidate predicates that were created previously, which in turn results in less candidate queries that need to be executed in the final and most expensive step.

Our system currently supports the following ranking criteria: $avg(A)$, $max(A)$, $sum(A)$, $sum(A + B)$, $sum(A * B)$, and no aggregation. Identifying candidate queries with $sum(A + B)$ ranking criteria is shown in Algorithm 9. For each numerical column pair (A_i, A_j) and distinct tuple ids set $\mathcal{I}_i$, the sum of the column values is aggregated, grouped by entity, and added to a list $Q(R')$ (Lines 1–6 in Algorithm 9). Thus, if the sorted list $Q(R')$ has a Footrule distance within the threshold θ, then queries formed by combining this predicate(s) with $sum(A + B)$ as a ranking criteria are identified as candidate queries.

Moreover, this approach can be generalized to finding any type of ranking criteria. The adjustment needs to be done regarding the different aggregation, arithmetical operation, or number of columns. One can look at these approaches as executing a query with different ranking criteria on R', since a predicate selects the tuples with tuple ids in $\mathcal{I}_i$. The number of tuple sets is lower than the number of predicates. Furthermore, R' is kept in-memory and implemented as a column-store, which makes this validations very efficient.

4.6.4 Ranking of Candidate Queries

The queries that qualify as candidates and whose results on R' have a Footrule distance within the user threshold θ need to be executed on the base table R. The execution is done in ascending order of Footrule distance, i.e., the candidate queries with results most similar to the input list L are executed first. This allows for early identification of similar queries as shown in the evaluation section.

4.7 Experimental Evaluation

We have implemented the approach described above in Java. Experiments are conducted on a 2× Intel Xeon 6-core machine, 256GB RAM, running Debian

	TPC-H	SSB
# Tuples	5313609	6001171
# Entities	171753	20000
# Textual columns	27	28
# Non-key numerical columns	13	20
# Avg tuples per entity	31	300
Highest # tuples per entity	187	579

Table 4.7: Table R characteristics

as an operating system, using Oracle JVM 1.7.0_45 as the Java VM (limited to 20GB memory). The base relation R is stored in a PostreSQL 9.0 database, with a B+ tree index on R's entity column.

4.7.1 Datasets and Workload

Datasets. We evaluate our approach of computing instance-equivalent and queries that produce similar entity rankings using data and queries of two benchmarks, **TPC-H** [TPC] and the **SSB** [PO]. For this, we created a scale factor 1 instance of both TPC-H and SSB data and materialized a single table R by joining all tables from their respective schema. The table R results in 57 and 60 columns, for TPC-H and SSB, respectively. The column *c_name* (from the customer table) acts as the entity column. We obtain tables with the characteristics described in Table 4.7.

We examine the applicability of our approach with variation of data in R, by performing experiments with sampling. As described in Section 4.5, operating on a sample of R has similar characteristics as working on a table R with modified data.

Queries. There are 13 and 22 queries available in the TPC-H and SSB benchmark, respectively. We adjusted the original queries by creating different query types ($max(A)$, $avg(A)$, $sum(A)$, $sum(A + B)$, $sum(A * B)$, and no aggregation), supported by PALEO (cf., Figure 4.4). We only write the ranking criteria when discussing the different query types. In order to examine the effects of the predicate size and selectivity factor, in each query, we vary the predicate size $|P|$, with $|P| \in \{1, 2, 3\}$. Queries with larger predicates have higher selectivity. Furthermore, all queries have the column *c_name* as an entity column. Example queries and their selectivity are shown in Table 4.8.

We execute each query Q over the table R to produce the top-k lists L. Using the LIMIT clause, we create top-k lists with $k \in \{5, 10, 20, 50, 100\}$. Then, we execute PALEO with inputs L and the table R. For the experiments involving sampling, we perform the experiments three times for each input list L and report on the **median** performance. In order to examine the effects of different sample sizes, we created experiments with sample size of 5%, 10%, 20%, and 30%. We keep the top-1000 entities for each numerical column. Furthermore,

	Query	sel.
TPC-H	$\gamma_{c\_name,MAX(o\_totalprice)}$ $(\sigma_{p\_type='MEDIUM\ POLISHED\ STEEL'}$ $\wedge\ r\_name='AMERICA'(R))$	0.001
	$\gamma_{c\_name,SUM(ps\_supplycost+ps\_availqty)}$ $(\sigma_{n\_name='JAPAN'}$ $\wedge\ p\_container='JUMBO\ BAG'$ $\wedge\ l\_shipmode='TRUCK'(R))$	0.0001
SSB	$\gamma_{c\_name,AVG(lo\_revenue)}$ $(\sigma_{s\_nation='UNITED\ STATES'}$ $\wedge\ p\_category='MFGR\#14'(R))$	0.002
	$\gamma_{c\_name,SUM(lo\_extendedprice*lo\_discount)}$ $(\sigma_{p\_brand='MFGR\#2221'}$ $\wedge\ s\_region='ASIA'$ $\wedge\ d\_year=1995(R))$	0.00003

Table 4.8: Example queries and their selectivity

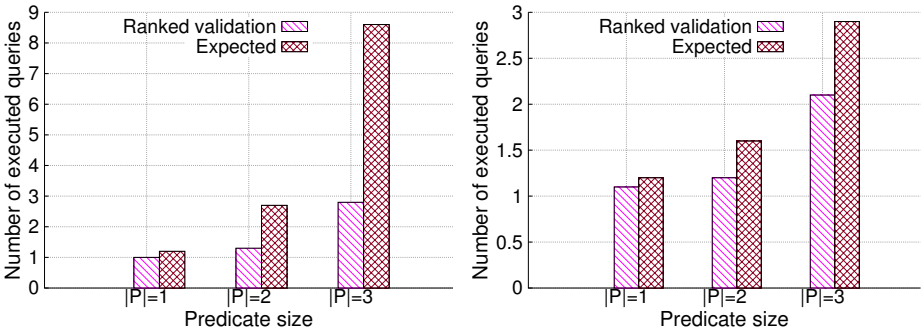
in evaluating our approach for computing similar entity rankings, to study the effect of θ to the system, we perform experiments with $\theta \in \{0.03, 0.06, 0.09\}$. We work with small values of θ since it is reasonable to assume that the user would only like to see queries similar to the input list and would like to avoid being overwhelmed with non-similar results. Thus, for each of these top-k lists we generated top-k lists within distance θ of the original one, by either randomly swapping two elements or by introducing a new random entity not in R. Then, we used these lists for the experiments with different values of θ accordingly.

Using the B+ tree on R, for each input list we retrieve (a sample of) R' and store it in memory. Thus, identifying the candidate predicates and ranking criteria are in-memory processes. Without using any compression techniques, the memory consumption of R' in our experiments was around 500MB. The query validation step is done by issuing queries to the underlying PostgreSQL database that resides on disk. Finally, queries show similar results depending on the number of columns in the aggregate function. Thus, for the sake of brevity, we discuss the results of $max(A)$ queries as representative of single column queries and $sum(A + B)$ for the two column queries. Finally, although PALEO discovers all valid queries for an input list, we focus on the efficiency of discovering the first valid query in the presented results.

4.7.2 Instance-equivalent Query Discovery

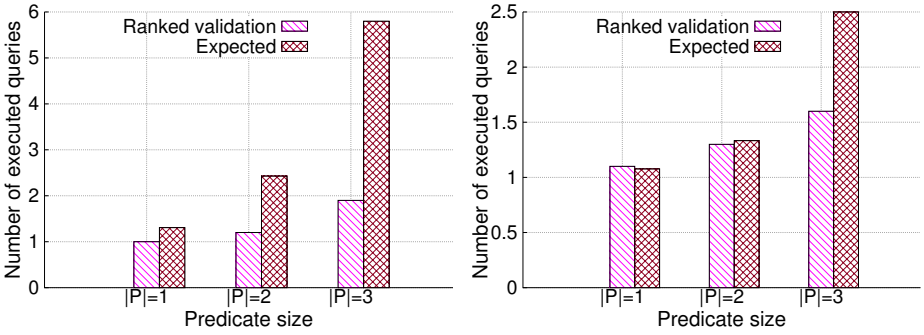
Valid Query Discovery. PALEO **always** discovers all valid queries for any of the supported query types when having available the entire table R'. The availability of all tuples ensures that false negatives are avoided, and introduces only (a small number of) false positives.

We observe that with all tuples from R' available, our system requires very



(a) $max(A)$ (b) $sum(A + B)$

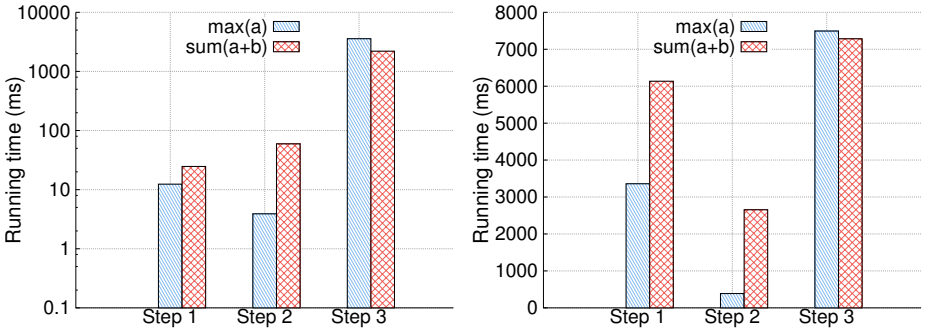
Figure 4.5: Number of query executions until first valid query with all tuples for TPC-H dataset



(a) $max(A)$ (b) $sum(A + B)$

Figure 4.6: Number of query executions until first valid query with all tuples for SSB dataset

few query executions in order to identify a valid query. Thus, for $sum(A + B)$ queries and the TCP-H dataset, the average number of query validations amounts to only 1.1 for $|P| = 1$, 1.3 for $|P| = 2$, and 2.1 for $|P| = 3$. In fact, for both TPC-H and SSB, **only a single query validation** is required for 76% of the top-k lists that stem from $sum(A + B)$ queries, while only two query executions are required for 14% of the top-k lists. Similarly, 65% and 70% of the top-k lists from $max(A)$ queries are found after a single candidate query is executed, while 26.6% and 16% after two query executions, for TPC-H and SSB respectively. Moreover, as shown in Figures 4.5 and 4.6, ranked validation outperforms the expected unordered validation and the benefit increases with predicate size. Note that when the entire table R is available, the candidate queries are executed in order of predicate selectivity, with more selective queries executed first. Intuitively, more selective queries decrease the probability of selecting other entities in the top-k part of the result, thus avoid false positives. The expected number of query validations reflects the case of executing candidate queries in random order. Assuming a uniform probability of the location



(a) TPC-H (b) SSB

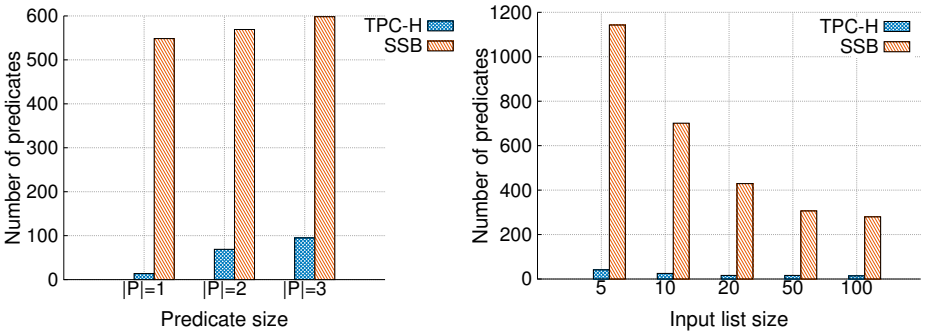
Figure 4.7: Running times by step

of the valid queries in the candidate list, we compute the number of expected validations with dividing the number of candidate queries with the number of valid queries.

Query Discovery Efficiency. We study the efficiency of the different steps from our system. Figure 4.7 shows the runtime of each step of our approach. As expected, the total runtime is dominated by the database-related operation, i.e., the candidate query validation (Step 3). Note that Figure 4.7 shows the runtime of finding the first valid query. We observe that for the TPC-H dataset the runtime of Step 3 is orders of magnitude higher than that of Step 1 and 2. Thus, for *max* queries the average runtime of candidate query validation is 3.6 *seconds*, while the average runtime of identifying candidate predicates and ranking criteria is 12.4 and 3.9 *milliseconds*, respectively. With the SSB dataset and the same type of queries, Step 3 needs 7.5 seconds, while the runtime of Step 1 and 2 amounts to 3.3 and 0.3 seconds respectively. The table R from the SSB dataset has more tuples per entity, which leads to having a larger R' and more data to process with our algorithms.

Identifying Candidate Predicates. We study the effect of predicate size and the length of the input top-k lists on creating candidate predicates. Figure 4.8 shows the number of created candidate predicates with different predicate and input list size. We observe that for the TPC-H data, the average number of created candidate predicates increases from 13.8 with $|P| = 1$, to 69 with $|P| = 2$, and to 95 with $|P| = 3$. We observe the same trend with the SSB dataset. Larger predicate size leads to generating more candidate predicates. The reason for this is that for a valid predicate with size $|P|$ we create as candidate predicates all sub-predicates with size smaller than $|P|$ as well. The number of shared tuple sets is smaller than the one of created predicates.

Figure 4.8(b) shows the average number of created predicates with different length of the top-k input lists. We observe that the number of candidate pred-



(a) By $|P|$ (b) By k

Figure 4.8: Number of candidate predicates for $max(A)$ queries

icates decreases with larger k. For TPC-H, the number of created predicates decreases from 41.3 for $k = 5$ to 14.3 for $k = 100$. With SSB, the average number of candidate predicates decreases from 1142.9 for $k = 5$ to 279.7 for $k = 100$. Larger k reduces the number of false positives in the candidate predicates. A predicate needs to select tuples with the distinct entities from the input list in order to qualify as a candidate predicate. With larger lists the number of entities increases, thus making it more difficult for a predicate to qualify as a candidate. Furthermore, we observe that a significantly larger number of predicates is created with the SSB data. This is due to the characteristics of the dataset, with SSB having more tuples per entity and more variety in data.

4.7.3 Evaluation with Sampling

The TPC-H generator creates uniform column distributions, thus the generated instance does not contain enough tuples per entity, with 14 tuples for an entity, at most. The SSB data has many tuples per entity, however these are extremely diverse in terms of predicates, i.e., the predicates found in the SSB queries often cover only a single tuple per entity. We thus focus on TPC-H data when employing sampling. There, for each tuple t in R we add n additional tuples, where n is a random number following the Gaussian distribution $\mathcal{N}(200, 50)$. These n tuples have the same values in the textual columns as t, but with non-key numerical values: $v = v + v \times abs(m)$, where $m \in [0, 1]$ is a random number following $\mathcal{N}(0.5, 0.5)$.

We study the effect of the sample size on the successful discovery of valid queries. We observe that a valid query is successfully discovered for **all** top-k lists that stem from single column queries, regardless of sample and predicate size. Figure 4.9 shows results for the discovery of $sum(A+B)$ queries. The discovery of valid queries depends on both the sample and predicate size. Having larger sample size enables better query discovery. For $|P| = 2$ and a sample size of 5% our system successfully manages to discover a valid query for 70% of the top-k lists. With a sample size of 10% the percentage of discovered queries increases

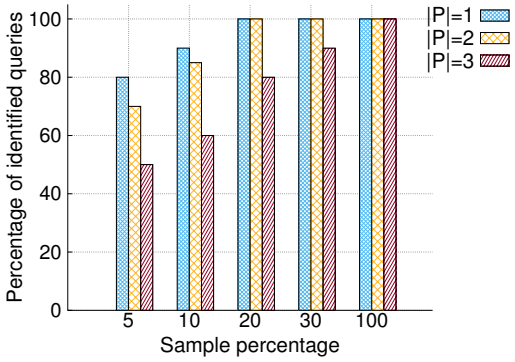
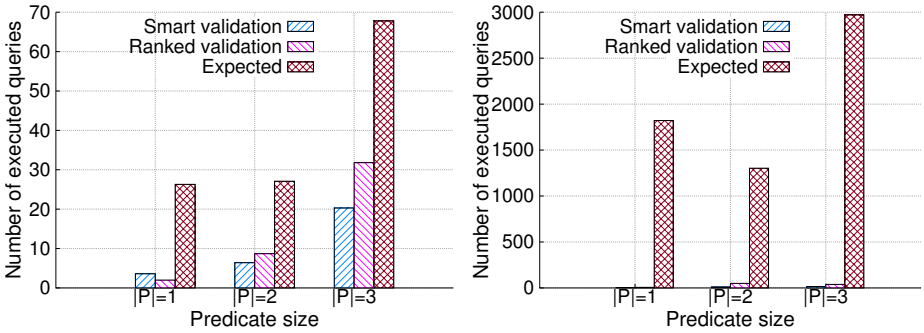


Figure 4.9: Valid query discovery with $sum(A + B)$ queries



(a) $max(A)$ (b) $sum(A + B)$

Figure 4.10: Number of query executions until first valid query with 30% sample for TPC-H data

to 85%, while with a sample size of 20% and larger, we manage to discover 100% of the queries with $|P| = 2$. Furthermore, we observe that discovering queries with larger predicate size is more difficult. With a sample size of 10% we successfully discover a valid query for 90% of the top-k lists with $|P| = 1$, 85% with $|P| = 2$, and 60% with $|P| = 3$. Queries with larger predicates are very selective, hence the probability of sampling tuples with a valid predicate is lower, which leads to false negatives. Sampling more tuples for these queries mediates this problem.

Smart Query Validation. Validating the created candidate queries is the bottleneck of our approach; executing (aggregated) queries on the database is expensive. We study the effects of our candidate query validation in terms of the computed query suitability and our result driven optimization. In addition, we investigate the effects of the predicate and sample size. Table 4.9 shows the average number of query executions needed using the two approaches for candidate query validation: smart result driven validation and ranked validation by query suitability. Furthermore, it shows the average number of created can-

		$max(A)$				$sum(A+B)$			
|P|	**Sample %**	**Smart**	**Ranked**	**# candidates**	**#valid Q**	**Smart**	**Ranked**	**# candidates**	**# valid Q**
1	5	20.6	24.6	163.7		16.6	32.1	11621.9	
1	10	13.7	12.4	185.1		24.9	28.2	10919.9	
1	20	5.1	3.4	144.7		9.8	16.2	10330.7	
1	30	3.6	2.0	105.1		4.3	6.6	7287.4	
1	100	1.0	**1.0**	4.8	4.0	1.1	**1.1**	4.8	4.0
2	5	**33.1**	69.8	161.3		**100.9**	1379.0	6540.4	
2	10	**23.4**	40.4	219.3		**47.4**	958.4	6991.2	
2	20	9.3	12.8	155.5		**20.4**	362.8	6605.4	
2	30	6.4	8.7	130.1		10.5	49.4	4820.4	
2	100	1.3	**1.3**	12.9	4.8	1.2	**1.2**	5.8	3.7
3	5	**59.4**	121.0	219.4		**199.0**	2510.6	3802.5	
3	10	**49.5**	129.4	282.0		**133.8**	982.4	4524.0	
3	20	**24.8**	56.4	224.0		**22.7**	61.4	3263.0	
3	30	20.3	31.8	203.5		15.4	38.5	4457.1	
3	100	2.8	**2.8**	25.7	3.0	2.1	**2.1**	4.4	1.5

Table 4.9: Number of candidate query validations with the different approaches by sample and predicate size for $max(A)$ and $sum(A+B)$ queries

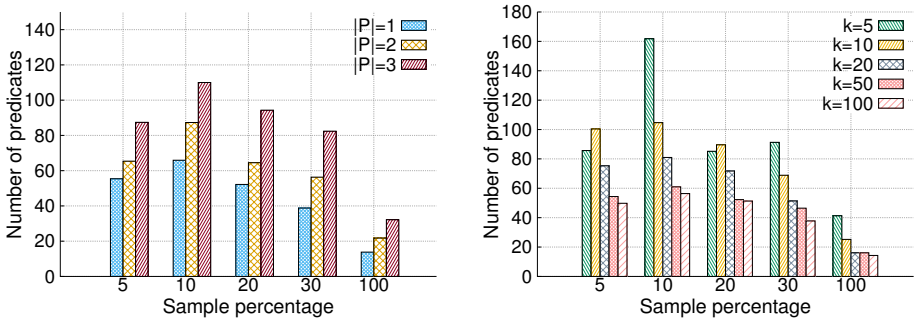
didate queries Q_c for each query type and the average number of valid queries identified when having all tuples from R' available.

Figure 4.10 compares average number of executed queries with our two approaches to validation with the expected number of query validations if the candidate queries are not ordered. For $max(A)$ queries, we observe that smart validation outperforms unordered validation by a factor of 7.3 with $|P| = 1$, 4.2 with $|P| = 2$, and 3.3 with $|P| = 3$. Furthermore, smart validation performs 26% query executions less than ranked validation with $|P| = 2$ and 33% less executions for $|P| = 2$. The benefits with discovering $sum(A + B)$ queries are even greater. Thus, smart validation in average reduces the number of expected query executions by a factor of 424.7 with $|P| = 1$, 124.7 with $|P| = 2$, and 192.6 with $|P| = 3$. The greater benefit with this type of queries stems from the fact that identifying the ranking criteria involves different combinations of columns, which significantly increases the number of candidate queries.

Furthermore, smart validation significantly outperforms ranked validation with smaller sample size. Thus, with sample size of 5% smart validation reduces the number of query executions for discovering $sum(A + B)$ queries over the rank-based validation by a factor of 13.7 and 12.6, with $|P| = 2$ and $|P| = 3$ respectively. Similarly, with a sample size of 10% smart validation reduces the average number of executions by a factor of 20.2 with $|P| = 2$ and 7.3 with $|P| = 3$. We observe that smart candidate query validation improves over rank-based validation for $max(A)$ queries as well, albeit with smaller but still significant effect. The greater benefit with $sum(A + B)$ queries stems from the fact that identifying the ranking criteria is more complex with this type of queries, thus making the query suitability less precise.

Larger sample size improves the candidate query suitability and reduces the number of candidate queries, thus resulting in less query validations. Smart validation reduces the number of validations for discovering $max(A)$ queries with a sample size of 30% by an average factor of 4.6 over a sample of 5%. Less candidate queries are created with larger sample size, since the availability of more tuples leads to better generation of candidate predicates and we discuss this later using Figure 4.11. Larger sample size significantly improves the approximation in finding the ranking criteria with $sum(A + B)$ queries and the factor of improvement amounts to 8.8 for the same sample sizes.

Larger predicate size increases the number of needed query validations. We observe that with a sample size of 30% discovering $max(A)$ queries requires 3.6 candidate query validations with $|P| = 1$, 6.4 with $|P| = 2$, and 20.3 with $|P| = 3$. With the same sample size, discovering $sum(A + B)$ queries needs 4.3, 10.5, and 15.4 query executions, with $|P| = 1$, $|P| = 2$, and $|P| = 3$ respectively. Queries with larger predicates are more selective, thus it is less probable that tuples selected by the valid predicate will be sampled. Additionally, subpredicates of a larger valid predicate can select the same tuples as the larger predicate, but in turn the smaller predicates are less selective which reduces their probability of being a false positive. Hence, candidate queries with smaller predicates can



(a) By $|P|$ (b) By k

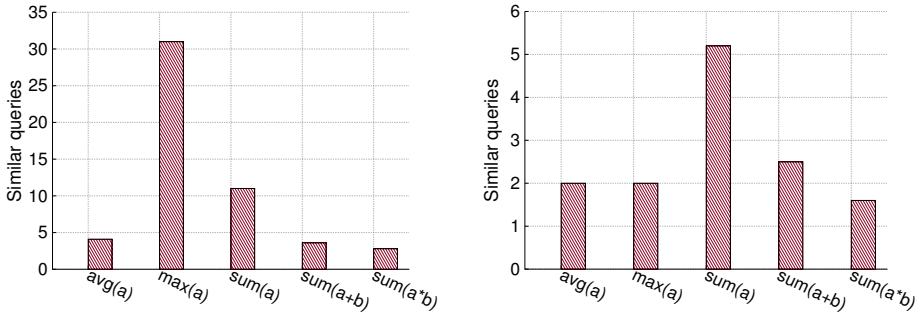
Figure 4.11: Number of candidate predicates for $max(A)$ queries

have higher query suitability.

Note that with sampling, the number of candidate queries for $max(A)$ queries is significantly lower than that of $sum(A + B)$ queries, as shown in Table 4.9. With single column queries identifying the ranking criteria is an easier task and we can limit the number of columns to consider as candidates. With $sum(A + B)$ queries on the other hand, the task of finding the ranking criteria involves combinations of two columns, thus making it more complicated. Furthermore, it is difficult to limit the number of column combinations to consider since a certain column with very large numbers (e.g., *total_price* in TPC-H) can dominate the sum. Hence, we consider all possible column combinations as candidate ranking criteria and rank them according to their approximated L1 distance.

Identifying Candidate Predicates. We study the effect of sample size on the number of created candidate predicates. We observe that the number of candidate predicates decreases with larger sample size. Larger sample size increases the probability of sampling larger number of tuples with a valid predicate, which in turn allows for stricter criteria in qualifying a predicate as candidate. Following the sampling probability in Section 4.5, with larger sample size we increased the ratio of covered entities in order to denote a predicate as a candidate. Thus, for sample size of 5%, the ratio of covered entities was set to 0.5, for 10% to 0.6, for 20% to 0.7, and to 0.8 for a sample size of 30%. Lower ratio avoids false negatives, but comes at the cost of increasing the number of false positives, since more predicates will qualify as candidates.

It is important to note that the experiments with sampling introduced expected variability. Depending on which tuples are sampled, the probability of the candidate predicates varies. Furthermore identifying the ranking criteria with $sum(A + B)$ queries is influenced by the sampled tuples. **Example:** We ran five executions of the input list from the second query in Table 4.8 with $k = 10$ and sample of 5%. As a best case a valid query is found after only 2 query execution, while 125 executed candidate queries were needed in the worst



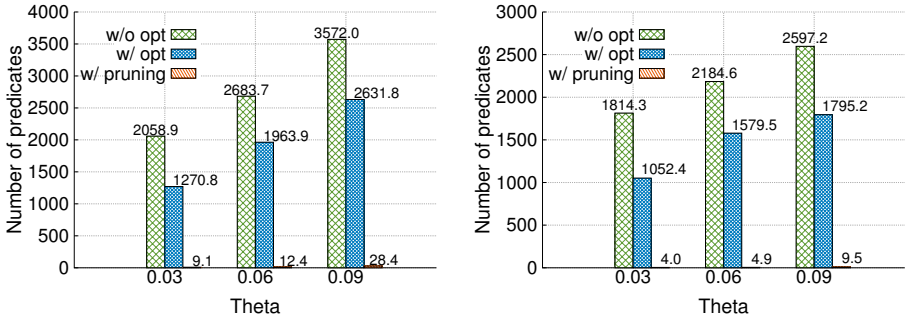
(a) TPC-H (b) SSB

Figure 4.12: Number of similar queries discovered with different ranking criteria

case. In the first case, the sampled rows contain the correct predicate for each distinct entity, i.e., the predicate probability is 1.0. Additionally, the correct ranking criteria (column combination) has the second lowest L1 distance on the valid predicate. It seems that the sampled rows were good for approximating the ranking values for the correct columns. In the other case, the predicate probability is 0.84 (14th in the ranking), while the correct column combination has a very large L1 distance, since the sampled tuples were not a good approximation of the ranking values. The remaining executions resulted in 27, 16, and 39 query validations. With smaller sample it is more difficult to find the ranking criteria for $sum(A + B)$ queries. This is a consequence of the non-uniform distribution of the values in A and B. Thus, the approximation depends on which tuples are sampled. Larger sample size mediates this problem. Having more tuples avoids the dependence on which tuples are sampled and leads to a more precise approximation.

4.7.4 Evaluation of Computing Similar Entity Rankings

Number of Similar Queries Discovered. Our system always manages to discover similar queries. Figure 4.12 reports on the average number of similar queries returned by the system over different values of θ and k, grouped by different query type, i.e., by the ranking criteria used for generating the input top-k list. We can see that for all the different query types, for both datasets, the system was able to find not only the original query that was used to generate the input list, but also several variations to it. With the TPC-H dataset the average number of similar queries was largest for those with $max(a)$ as a ranking function. Taking a deeper look shows that this was due to the selection part of the query. Many queries with the same aggregate function ($max(a)$) but with different predicates produced results with Footrule distance within θ. This is intuitive for this type of queries, if the tuple that is in the top-k result has a valid predicate of size n, all sub-predicates with size smaller than n will also be



(a) TPC-H (b) SSB

Figure 4.13: Number of candidate predicates

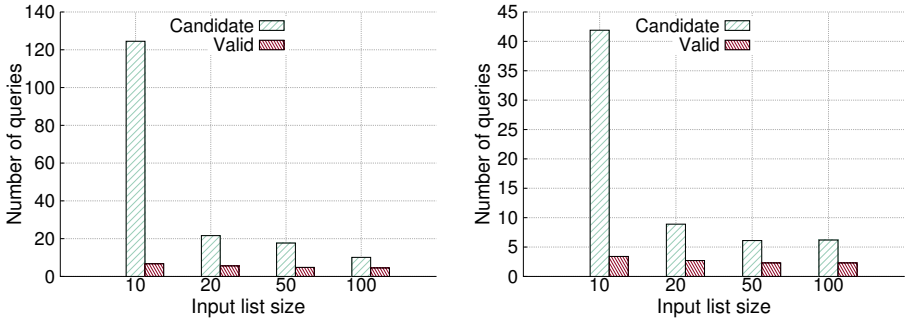
in the top-k result. This is not the case with the SSB dataset, it seems that the data there does not have these characteristics. Thus, the largest number of similar queries for this dataset was returned for the top-k lists ranked over the sum of one attribute ($sum(a)$). This is because many different ranking criteria, combined with the same predicates, produce similar top-k lists. The sum over a single column is susceptible to domination of certain columns. Thus, if the original query was ranked with sum over a column with large values, an aggregation with arithmetic operations that involve this column and columns with smaller values would not change the order in the result too much.

Our system is not only effective in finding the similar queries, but is also very efficient. Table 4.10 reports on the **precision and recall at 10**, i.e., the ratio of the number of results in the first 10 executed candidate queries, and the ratio of the number of results found after executing the first 10 candidate queries and the total number of results. We can see that for the SSB dataset the recall@10 is in all cases larger than 80%, meaning that we were able to find more than 80% of the similar queries by only executing 10 candidate queries. For the TPC-H dataset, the recall is slightly lower, however, in average, we can still find more than 50% of the similar queries by executing 10 candidate queries. We can see that even when we have a large number of candidate queries (117.6 for queries ranked by $sum(A)$ and $\theta = 0.09$), the system finds more than 60% of the results within the first 10 query executions. For this dataset the precision is higher meaning that within the 10 queries executed, larger percentage are similar queries. For the SSB dataset the precision@10 is lower but this is simply due to the lower number of valid similar queries.

Number of Candidate Predicates. Figure 4.13 reports on the number of candidate predicates created with and without the optimizations proposed in Section 4.6.2, and after pruning out predicates as discussed in Section 4.6.3. We see that for both datasets the system was able to prune out more than 99% of the candidate predicates without any query execution over the database. This shows that our algorithms are very efficient. First, the optimizations of knowing which

Query type	θ	TPC-H				SSB			
		# candidate Q	#valid Q	P@10	R@10	# candidate Q	#valid Q	P@10	R@10
$avg(A)$	0.03	60.5	4.1	0.31	0.53	9.5	2.0	0.57	0.82
	0.06	64.0	4.1	0.30	0.53	9.3	2.0	0.55	0.81
	0.09	73.5	4.1	0.27	0.51	14.4	2.0	0.47	0.81
$max(A)$	0.03	136.9	31.7	0.49	0.53	7.8	2.0	0.49	0.92
	0.06	486.0	31.7	0.43	0.54	10.2	2.0	0.46	0.88
	0.09	484.9	29.7	0.42	0.53	25.0	2.0	0.39	0.85
$sum(A)$	0.03	16.0	10.9	0.75	0.75	10.6	5.1	0.72	0.90
	0.06	22.7	11.1	0.66	0.70	15.7	5.2	0.60	0.85
	0.09	117.6	11.1	0.56	0.64	33.5	5.2	0.53	0.80
$sum(A+B)$	0.03	6.1	3.4	0.80	0.95	6.2	2.5	0.69	0.95
	0.06	18.3	3.3	0.60	0.83	11.8	2.2	0.51	0.94
	0.09	82.4	4.0	0.50	0.80	46.2	2.9	0.37	0.84
$sum(A*B)$	0.03	4.2	2.8	0.85	0.97	2.7	1.6	0.76	1.00
	0.06	9.2	2.8	0.80	0.86	4.5	1.6	0.64	0.93
	0.09	47.2	2.8	0.72	0.87	28.7	1.6	0.54	0.92

Table 4.10: Number of candidate and valid queries, precision, and recall for the different query types for different θ



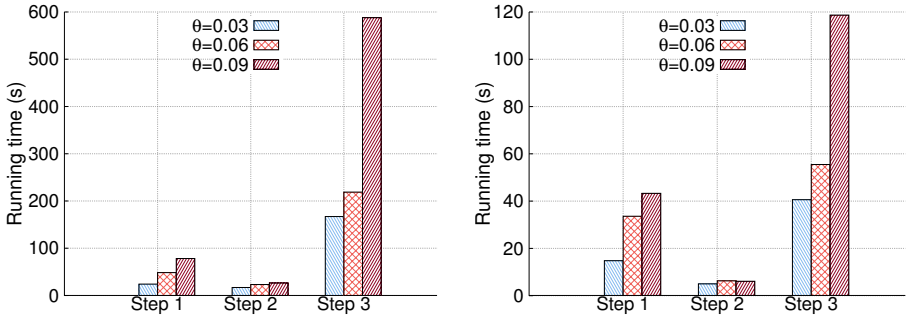
(a) TPC-H (b) SSB

Figure 4.14: Number of candidate and valid queries for different k

subset of the elements is needed for generating the candidate predicates, the system was able to filter out at least 27% of the candidate predicates, considering all three values of θ, for the SSB dataset, and at least 24% of the data for the TPC-H dataset. In general, the percentage of filtered predicates with this method reduces as θ increases, since the number of mandatory entities that we can use reduces. Furthermore, by joining the candidate predicates with a ranking criteria, and then comparing the possible result over R' with the input list, we were able to further reduce the number of candidate predicates. In addition, we see that as θ increases the number of candidate predicates increases as well. This is due to the more relaxed conditions which qualify a predicate as a candidate with larger θ, as well as the fact that results with larger Footrule distance would also qualify as candidates.

Figure 4.14 shows the average number of candidate queries executed over the database, compared to the number of valid queries, for different values of k, for both datasets. Note that as valid queries we refer to those queries whose resulting top-k lists R_L is within Footrule distance θ to the input list L, i.e., $F(R_L, L) \leq \theta$. We see that, except for $k = 10$ the number of candidate queries that needs to be executed amounts to at most 4 times the number of valid queries for both datasets. However, this number decreases even further as we increase k since we have more information about the input list.

Runtime of the System. Figure 4.15 reports on the average execution time of the three stages of the system for different values of θ. It shows the average runtime per input list for identifying all valid queries. The results for both datasets are similar. As expected, the execution of the candidate queries is the most demanding part of the system, since the queries are executed over a database that resides on disk. Finding the candidate predicates is less efficient than finding the ranking criteria, since even with knowing the subset of entities that must be satisfied by the predicates, the set of possible candidate predicates is larger than the set of possible ranking criteria. In addition, we can see that for Step 1 the execution time increases with θ linearly, which is understandable, since the number of predicates also increases with θ, because we are not able to



(a) TPC-H (b) SSB

Figure 4.15: Runtime by step

prune as many predicates. On the other hand, we see that finding the number of candidate rankings is almost unaffected by θ. The execution time of Step 3, executing the queries and validating them, is most affected by increasing θ. Larger θ introduces more variability, which allows creating more candidate queries.

4.7.5 Lessons Learned

In summary, the main findings from the above experimental results are:

- Our system always discovers a valid query with all tuples from R' available.
- Finding a valid query is done efficiently for both datasets and requires just a few query executions with only a single query validation for 76% and 68% of the top-k lists that stem from $sum(A + B)$ and $max(A)$ queries, respectively.
- Sampling introduces the possibility of false negatives. However, we manage to discover a valid query for all top-k lists that result from a single column query. Finding valid $sum(A + B)$ queries is more difficult and we manage to identify a valid query for 96.7% of the top-k lists with a sample size of 30%.
- Identifying the candidate predicates and ranking criteria is done in-memory and are very efficient. Validating candidate queries is done by executing them on an on-disk database, thus having the majority of the total runtime.
- The smart result driven candidate query validation significantly reduces the number of query executions needed in finding a valid query. In addition, larger predicate size leads to more query validations.
- Larger sample size reduces both the number of false positives and false negatives in the candidate predicates.

- Having more data improves the ranking of the candidate ranking criteria, since we have better approximation of the L1 distance.
- In computing similar entity rankings, the system efficiently discovers all similar queries. With the SSB dataset, the recall@10 for the identified similar queries is in all cases larger than 80%, while with TPC-H we get recall@10 is always larger than 50%.
- The pruning optimizations for discovering similar queries, result in pruning more than 99% of the candidate predicates without any query execution over the database.

4.8 Use-case Applications of PALEO

This section is based on our demo publication at VLDB 2016 [PMMP16]. We created a full-fledged prototype centered around our PALEO framework for reverse engineering OLAP-style database queries, described in detail in the previous sections of this chapter. Users can post lists of entities for which explanatory SQL queries and full result lists are returned. By refining the input, the results, or the queries, users can interactively explore the database content.

To demonstrate our system and all its capabilities to users we propose the use of three datasets. The first dataset contains basketball statistics from the Database Basketball portal [Dat]. It contains per-season statistics of players and their teams, capturing the NBA and the ABA leagues from 1946 to 2009. In total, there are 3924 players and 95 teams. The database reports on various numerical attributes such as turnovers, rebounds, total points per game, assists, field goals, etc. Furthermore, we also use a subset of the IMDb dataset [Int], which contains info on movies, actors, genres, movie length, ratings, etc. The third dataset is the DBLP [DBL] database containing information about authors and publications focused in the field of computer science.

We have specifically chosen datasets that allow creating a large variety of top-k rankings, with different categorical constraints and ranking functions. Moreover, the domains of the datasets are diverse enough to make the use of the system interesting for people with different interests. Following, we propose a few application scenarios for exploring data with PALEO.

4.8.1 Exploring Similar Lists

This application scenario shows, given a ranked list of entities and a similarity threshold θ, a set of queries together with their top-k result lists and corresponding statistics. Figure 4.16 shows a screenshot demonstrating this scenario. The user can specify the top-k list by entering entities through an input field and then possibly rearranging them (marked with the top rectangle in Figure 4.16). The system provides several interesting starting examples to assist users in getting started. PALEO provides assistance when entering input entities; alternatively

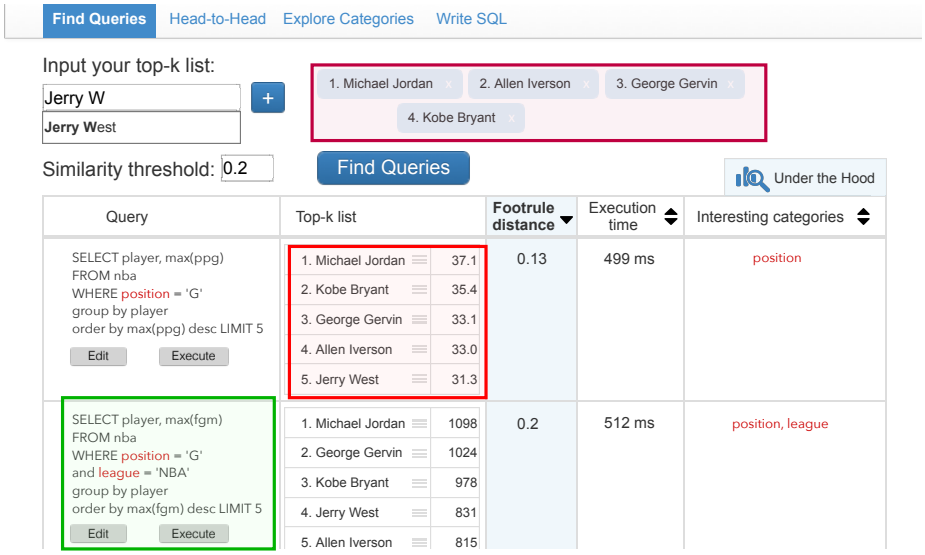


Figure 4.16: Screenshot of scenario for finding similar lists

SQL queries can be used to retrieve a ranking that can be altered by users to get reverse-engineered. Users need to enter a similarity threshold, i.e., the maximum Footrule distance the input list can have to the lists produced by the identified queries.

The system starts by pressing the "Find Queries" button. As output, the system shows to the user the Footrule distance of each query:list pair, the query execution time, and marks the interesting attributes in its predicate according to the classification of categorical attributes. The pairs with lower Footrule distance will be displayed more prominently by default, with the number of interesting categories as a second criteria. However, users can additionally change the ordering of the results: by the number of interesting categories in a query's predicate, or the query execution time. The result rankings can be re-ordered and used as input in a following execution, thus providing insight into the changes of the reverse engineered queries (marked with red rectangle in the middle of Figure 4.16). Moreover, the reverse engineered queries can be manually edited and executed to see the changes they produce in the rankings (marked with green rectangle in the bottom-left of Figure 4.16). By inspecting interesting categories, users can see which categories were used in the `WHERE` clause of the query. For instance, in the example in Figure 4.16, the upper query uses only a constraint on the field position while the query below puts constraints to position as well as league. Additional statistics of the entire process can be shown by pressing the "Under the Hood" button. The user can see the number of query executions needed until the first valid query is identified, the runtime of each of PALEO's steps, the number of candidate predicates that were created and pruned, and the number of created candidate queries.

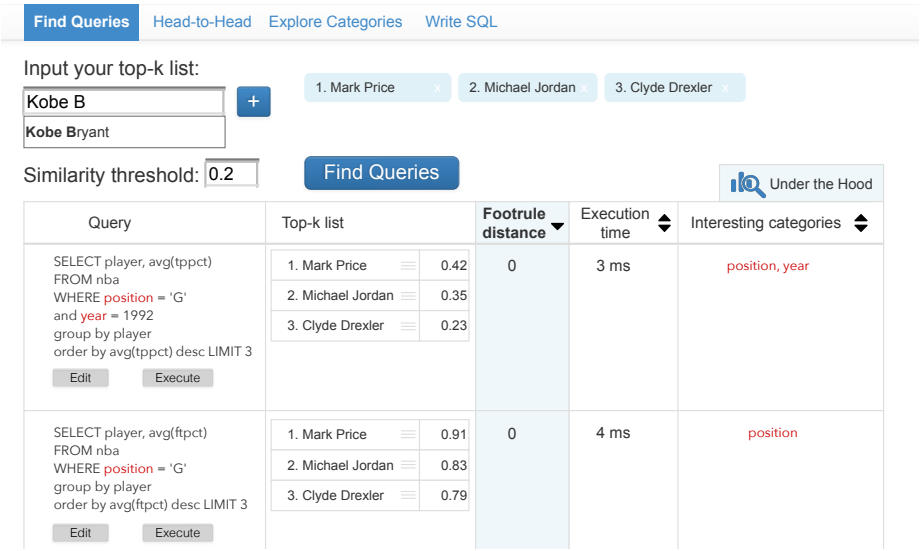


Figure 4.17: Screenshot of scenario for head-to-head comparison of entities

4.8.2 Head-to-head Comparison of Entities

In this application scenario, users can perform a comparison of specific entities of their choice. PALEO shows users how entities compare with each other based on different scoring functions and categories. In a head-to-head entity comparison, the performance of entities is agnostic to all other entities in the domain except the ones provided as input. Thus, the users are also able to compare entities that normally do not belong in the same league, i.e., one is ranked significantly higher than another and therefore are not together in any top-k scenario. PALEO still provides an option of exploring these entities and shows how they compare against each other by ignoring the other entities that could (potentially) appear between them in the ranking.

For instance, consider a scenario where the user wants to see how Mark Price compares with the best guards in the NBA, even though this player is never really compared to the top guards that ever played in the NBA. This is because he cannot be compared to the elite, when considering the points scored, the assists made, or the rebounds, etc. However, for example, the first query in Figure 4.16 can still be identified as a valid query, even though if considering all entities in the dataset, Mark Price is not in the top-100 players by points per game in a single season with 19.6 PPG. Furthermore, as shown in Figure 4.17, other interesting rankings can be found where Price is ranked even above Michael Jordan, e.g., for the free throw percentage or three point field goal percentage. In this way, the users can see the virtues of these non-elite players, i.e., the performance aspects where they can compete with the elite. In this scenario, PALEO only needs to utilize the table R' since it already contains all tuples for the entities in the input list. In this way, valid queries are identified very

efficiently. The user can also ask to see the entities that are ranked between or above the input entities for a specific query. For instance, she wants to see how far away is Mark Price from the top-100 player ranked by points per game. Our system will do that, by additionally executing the query over the database. Then, the entire top-k list will be shown to the user.

4.9 Summary

In this chapter, we described a framework to reverse engineer top-k OLAP queries. This has turned out a complex problem given the various dimensions of the search space, the potentially very large base relation, and the small input snippet in form of a top-k list. Our approach mainly operates on a subset of the base relation, held in memory, and further uses data samples, histograms, and simple descriptive statistics to identify potentially valid queries (that generate the input list). We proposed a probabilistic model that evaluates the suitability of a query discovered over a subset of R', methodology that is directly applicable to the case of handling variations of R. In any case, when trying to identify promising queries, the main difficulty is to limit the number of false positives—that cause unnecessary query validations—as well as to limit false negatives—that cause loss in recall. The ordering of potentially valid queries according to the probabilistic model in addition to an iterative refinement of the validation of candidate queries was proven to drastically decrease the amount of time to validate (or invalidate) queries in the final stage of the approach. This is specifically true for cases of low sampling rates—and expectedly likewise for significantly changed data. We extended the framework to be able to determine queries that generate similar ranked lists according to a user-specified similarity threshold and modified the algorithms by embedding distance-measure specific bounds into the query generation process. Finally, we presented several use-case application scenarios, demonstrating the usability of our framework in interactively exploring data.

Chapter 5

Reverse Engineering Top-k Join Queries

5.1 Introduction

In the previous chapter we described the details of reverse engineering top-k queries working on a single database relation. However, data in modern relational database systems and specifically in data warehouses is often stored in complex schemas containing many relations. In this chapter, we present an approach to identify top-k OLAP queries with arbitrary number of joins of multiple relations. Given a top-k input list, we focus on discovering the join predicates of the query. Previous work [ZEPS13, SCC+14, PDCC15, KLS18] considered the problem of reverse engineering join queries, nevertheless none of them support OLAP-style queries with aggregation functions nor are based on a ranked top-k input. We use the approach presented in [ZEPS13] as guidance, however, using a small top-k list as input and supporting aggregation functions change the problem significantly. The content of this chapter is based on our publication at BTW 2017 [PWM17].

Figure 5.1 shows a sample database containing sales data. It contains information about customers, such as where they come from, the orders they placed, what parts were bought in each order, similarly to TPC-H [TPC]. In this scenario, it is very common to join the relations to gain insight about the ordering patterns of customers. Consider the top-k list shown in Figure 5.2(a). It contains the customer name and the score that was used as ranking criteria according to which the customers are ordered. Considering the database in Figure 5.1, we can find that the top-k list can be generated using the query in Figure 5.2(b), joining the Customer and Orders relations. It computes the top-3 customers, living in the USA, ranked by the maximum price of an order that they made. There can be other queries that produce the same input list, joining a different set of tables, which enables further exploration of the underlying data. Reverse

Customer

CustID	Name	Country	Balance
1	Bruce Wayne	USA	250.49
2	Clark Kent	USA	124.56
50	Tony Stark	USA	45.99

Part

PartID	PName
1	LG G7
2	Galaxy S9
3	iPhone X

Orders

OrdID	CustID	Price	Date
1	1	199.99	28.11.14
2	1	749.90	01.04.15
23	2	199.99	30.12.12
93	50	1000.00	27.02.15

LineItem

ItemID	OrdID	PartID
1	1	1
2	2	2
15	23	1
18	50	3

Figure 5.1: Sample database with sales data

Tony Stark	1000.00
Bruce Wayne	749.90
Clark Kent	199.99

(a) Input list L

```
SELECT c.name, max(o.price)
FROM customer c, orders o
WHERE c.custid = o.custid
AND c.country = 'USA'
GROUP BY c.name
ORDER BY max(o.price) DESC
LIMIT 3
```

(b) Result query

Figure 5.2: Example input L and result query

engineering the join predicates of even this rather simple top-k query on the sample database is not trivial, as the information that can be derived from a two-dimensional top-k list is very limited, especially with small k.

5.1.1 Problem Statement

Given a database D containing relations $\{R_1, R_2, \ldots, R_t\}$ with schema $R_i = \{A_{i_1}, A_{i_2}, \ldots\}$, and an input list L that represents a ranked list of items with the numerical values that were used for the ranking. The system task is to efficiently and effectively discover join queries that, when executed over the database D, compute result lists that match the input L.

We identify top-k select-project-join queries of the form shown in Figure 5.3(a). This task can be broken down into three sub-tasks: identifying the join predicates J_i, finding the filter predicates and ranking criteria, and validating the queries. In this work, we specifically focus on finding the query graph, i.e., identifying the foreign key/primary key join constraints of the query. We abbreviate primary key as pk, and foreign key as fk in the remainder of this thesis.

Definition 9 Query Graph
A query graph is a graph $\mathcal{Q}$ with nodes corresponding to database tables in D and edges representing fk/pk constraints.

```
SELECT L.e, agg(value)
FROM R1, R2, ...
WHERE J1 and J2 and ...
    and P1 and P2 and ...
GROUP BY L.e
ORDER BY agg(value) DESC
LIMIT k
```

(a)

L

$L.e$	$L.v$
e	100
f	90
g	80
m	70
o	60

(b)

Figure 5.3: Join query template and example input list L

The task is to efficiently identify the join constraints of a valid query, i.e., one whose result matches L. This consists of determining the tables in the FROM clause of the query (e.g., customer c, orders o), as well as the fk/pk relationships therein (e.g., c.custid = o.custid). Identifying filtering predicates and ranking criteria was discussed in detail in Chapter 4 where the system operates on a single table R and joins were not considered.

The ranked list L, which serves as input for the system and resembles the desired query output, consists of two columns and k rows. The first column, denoted $L.e$, contains entity identifiers, while the second column, denoted $L.v$, contains possibly aggregated numerical values. The input is ordered by the second column, meaning the top-k entities are ranked based on their "score". L does not contain any information about the database columns containing the entities or the aggregated score values. An example input list is shown in Figure 5.3(b); the column names are omitted in the input as they need not generally be related to database column names.
In addition to finding the fk/pk join constraints J_i, we support predicates P of the form $P_1 \wedge P_2 \ldots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_i = v$ (e.g., country = 'USA').

5.1.2 Contributions and Organization

In this chapter, we present an approach that specifically deals with solving the task of reverse engineering top-k *join* queries with arbitrary number of fk/pk joins. With this work we make the following contributions:

- We present a framework which efficiently computes a list of join query candidates by making use of database meta-data and appropriate data structures.
- Additionally, we present a ranking and verification system which ensures fast execution of candidate join queries. Furthermore, queries with a high probability of being a match are tested first while keeping database interactions low.
- We introduce several optimizations which improve the running time of the framework, including a candidate classification system.

- We present the results of experimental evaluation conducted on a database containing TPC-H [TPC] benchmark data.

This chapter is organized as follows. Section 5.2 presents system overview. Section 5.3 establishes the key ideas of our approach to handling joins, followed by introducing certain optimizations in Section 5.4. Section 5.5 gives a brief overview of the remaining parts of our framework, while Section 5.6 reports on the results of the experimental evaluation and presents lessons learned. Section 5.7 summarizes the work in this chapter.

5.2 System Overview

The system for discovering top-k SPJ queries with arbitrary number of joins contains three main components. We see the three main components depicted in Figure 5.4.

- Identify the join predicate, i.e., what are the possible fk/pk constraints in the query
- Identify the filter predicates and ranking criteria
- Validate queries

The first component, resembling the contribution of this work and presented in green in Figure 5.4, consists of generating a ranked list of candidate joins by using mostly database meta-data and minimal access to actual table data.

Definition 10 Candidate Join
We say a query graph $\mathcal{Q}$ with a set of tables R_i and their fk/pk join predicates is a ***candidate join*** *iff projecting and selecting on the table R created from the join can result in the input list L. Formally,*

$$R = \bowtie (R_i \in \mathcal{Q}),\ \exists\ \textit{projection}\ \pi,\ \textit{selection}\ \sigma\ :\ \pi(\sigma(R)) = L$$

The table R created from a candidate join contains all the tuples from the input list L. Furthermore, it contains additional tuples that are not (yet) filtered out by a filter predicate and other columns that will not be used by the projection. Additionally, it contains multiple tuples per entity from which the ranking criterion needs to compute an aggregated score. Thus, the table R is a superset of the input list L and contains all the necessary data to produce the input list by applying the appropriate filter predicates and ranking criterion.

The second component takes as input the candidate join predicates and, for each created join table R, identifies possible filter predicates and ranking criteria. Then, by combining the join conditions, the filter predicates, and the

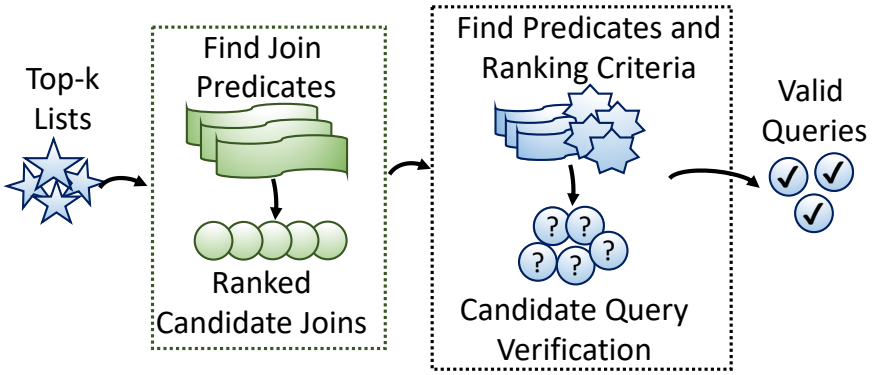


Figure 5.4: System architecture

ranking criteria, full-fledged candidate queries are created. The final component verifies these queries by executing them on the database and comparing their results with the input, those that match the input are returned as valid queries. The latter parts of the system resemble the PALEO framework presented in Chapter 4 and are briefly reviewed in Section 5.5. In this work, we devise an efficient approach in identifying the join predicates and consider techniques that could facilitate the next steps to determine the remaining parts of a valid query.

5.3 Finding Join Predicates

The task of identifying candidate join trees consists of the steps depicted in Figure 5.5. These steps are all part of the Find Join Predicates component, shown in the green part of Figure 5.4. The following sections will describe the six steps needed to generate the ranked candidate joins from the input L, which will then be processed by the remaining components of the framework.

5.3.1 Step 1: Schema Exploration

In the Schema Exploration step, the system tries to determine which column in the database schema was used as the entity column. Furthermore, by leveraging basic database metadata, it selects suitable columns as early candidates for the ranking criteria.

Identifying the correct entity column is a straight-forward task. Finding the column from which the input entities originate is done by using a B+ tree index, or alternatively, the entity column can be efficiently identified by implementing an auxiliary inverted index. Finding suitable ranking criteria, however, cannot be done by simple inclusion check of the values in L, since these may stem from an aggregation. Thus, candidate ranking columns are determined based on the data type of the columns.

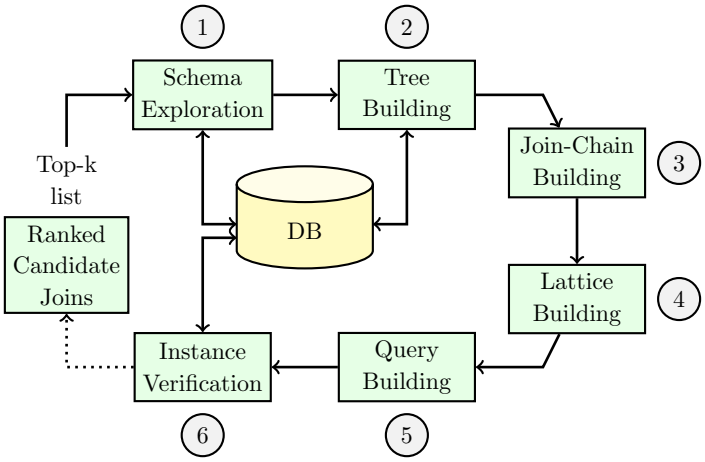


Figure 5.5: Generating the ranked candidate joins

5.3.2 Step 2: Building Join Trees

For each entity candidate column $R.e$, this step explores the database schema and tries to build a tree-like query graph that we call a join tree rooted at $R.e$. All candidate joins are extracted from such trees. We define a depth d of the tree that limits the complexity of the join candidates inspected by the following stages of the framework. A sufficiently high depth d guarantees the successful discovery of a matching query generating L. In turn, if d is chosen too high, the performance of the algorithm may suffer, as a large amount of overly complex candidate joins will be generated.

An example join tree of depth 2 is illustrated in Figure 5.6 following the TPC-H schema, with nodes corresponding to instances of database tables and edges relating to key-constraints between tables. Starting from the entity candidate table Customer as a root node, the child table Nation is referenced by a fk inside the Customer table, while the Orders child node contains a fk referencing the pk of the Customer relation. All tables related by fk/pk keys will be added as child nodes to the current node, until the maximum depth of the tree has been reached. Hence, each node except for the root node contains its parent node as a child node, unless it is a leaf of the tree. As this results in a lot of nodes instantiating the same database relation, a running index is appended to each node identifier, which now consists of the table name and a number (e.g., Customer1). Having multiple instances of the same table in a join can serve as a proper filter and removing one of them changes the output. Unique node identifiers are important for referencing nodes in later steps of the algorithm, especially when nodes are being merged. Additionally, all of the edges carry information about the table and column names involved in the join to ease the translation into SQL during the Query Building step. The process of building the tree does not have a large performance impact on the database, as the database metadata needs to be queried only once to gain information about all

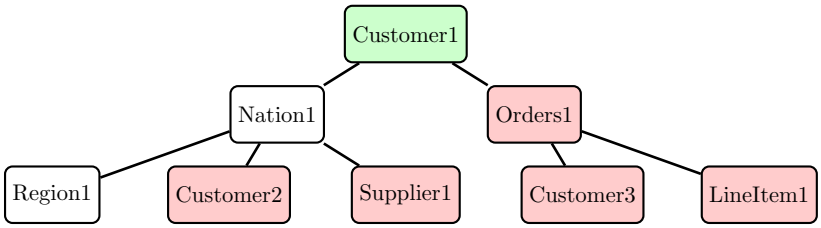


Figure 5.6: Example join tree of depth 2 with marked nodes

key-constraints in the relevant schema.

Marking nodes in join trees. After a tree is built, the algorithm continues with marking certain nodes. All ranking candidate columns which have been identified during Schema Exploration will be looked up in the tree and nodes originating from the corresponding tables will be marked as ranking candidate nodes $R.r$, which may also include the root of the tree. Figure 5.6 shows an example tree with ranking candidate nodes marked in red. The root node of the tree is both an entity and a ranking candidate and therefore marked with green in the figure. Marking the nodes helps to simplify the approach of constructing the join chains, which always have to include at least one marked node and the root of the tree.

5.3.3 Step 3: Building Join Chains

The goal of building the join chains and the lattices during the following step is to represent all possible query graphs connecting the entity table to a table containing a ranking candidate column. Hence, a join chain is a set of nodes from the join tree built in Step 2 which includes the entity node and at least one marked node.

Definition 11 Join Chain
A join chain $\mathcal{J}$ is a linear query graph that always contains the database table R_e which was identified as the entity table during Schema Exploration as one node of the graph and at least one node corresponding to a table R_r which contains a ranking candidate column. A join chain $\mathcal{J}$ generalizes a query graph $\mathcal{Q}$ if the following two conditions are true:

1) The set of tables corresponding to nodes in $\mathcal{Q}$ is equal to the set of tables corresponding to nodes in $\mathcal{J}$

2) Each node in $\mathcal{Q}$ can be mapped to a node in $\mathcal{J}$ and this mapping is an injective function relating nodes to nodes corresponding to the same database table.

To construct a join chain, the method begins with a random node in a join tree $\mathcal{T}$, which serves as the starting point of the join chain, and then navigates to its parent node, adding it to the chain in the process. This step is repeated until

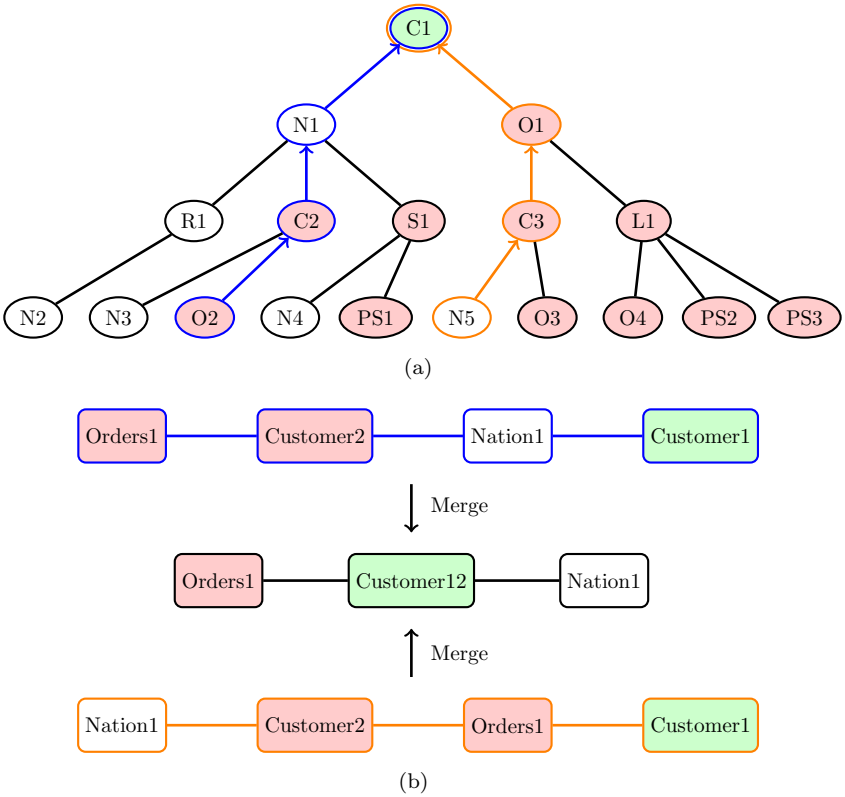


Figure 5.7: Example (a) join tree and (b) merging of nodes

the root of the tree has been reached, concluding the join chain construction by adding the entity node R_e. By consecutively using all individual nodes contained in the tree as a starting point for a join chain, all possible chains are eventually constructed.

Figure 5.7(a) shows an example join tree of depth 3. Two join chains are highlighted in the tree, one starting at the node O2, and the other starting at N5. Both join chains essentially contain the same set of tables: two instances of the Customer relation and one instance each of the Orders and Nation relation. Hence, these join chains only differ in the order of the join operations, the reason for this being the multiple instances of the Customer relation.

5.3.4 Step 4: Building Lattices

Join-chains are linear graphs connecting nodes of a tree to its root. The join chains generated during the previous step will serve as root nodes for the lattices built in this next step of our approach. By iteratively merging duplicate nodes, i.e., nodes stemming from the same database table, new query graphs are built, corresponding to vertices inside the lattices, as shown in Figure 5.8. Each edge in the lattice translates to one merge step. By performing these merge steps

over all join chains, all possible query graphs will be built eventually, meaning the query graph representing the query generating L will be among them if the depth of the computation was chosen sufficiently high.

Theorem 5.3.1 *If the data inside the database has not changed, then for any query graph $\mathcal{Q}$, there exists at least one join chain $\mathcal{J}$ that generalizes $\mathcal{Q}$, and $\mathcal{Q}$ is a node in the lattice rooted at $\mathcal{J}$.*

Proof Theorem 5.3.1 follows the work in [ZEPS13]. First, we prove that for any query graph Q there exists at least one Join-Chain J generalizing Q. Secondly, we show that Q is a node inside the lattice rooted at J.

Hypothesis Let Q be an arbitrary query graph with n nodes, then there exists a Join-Chain J that generalizes Q.

n=1: If Q consists only of one node N, then the table corresponding to this node $N.table$ contains the entity and score column. The Join-Chain generalizing Q consists only of one node N', the root of the tree T which was generated during Tree Building. $N.table = N'.table$ because the root node of T is the table containing the entity column by definition.

n=n+1: Let Q_n result from Q_{n+1} by removing one node N from the query graph, which is not the entity node. Then J_n is a Join-Chain generalizing Q_n by application of the hypothesis and T the corresponding tree. To construct a Join-Chain J_{n+1} which generalizes Q_{n+1}, we start with J_n. Remember that J_n represents a linear graph connecting a node E in T to the root of T. We now append the inverse of J_n minus the endpoint E itself to E, which is possible because key-constraints can be followed in both directions when constructing T, and we call the resulting Join-Chain J_{2n-1}. Starting at the new endpoint of J_{2n-1}, which is now corresponding to the root table of the tree, we can travel further down the tree T until we reach a node N' with $N.table = N'.table$. This has to be the case because N is reachable from some node in Q_n and every node in Q_n maps to one node in J_n, which can be reached from the root node. On the way to reach N' we visit x tables already present in J_n, resulting in a new Join-Chain J_{n+x}. Every node in J_{n+1} can be mapped to a node in J_{n+x} and although J_{n+x} contains many duplicate nodes, the set of tables of J_{n+1} and J_{n+x} is identical. Hence, J_{n+x} generalizes Q_{n+1}. ∎

Merging Nodes. Theorem 5.3.1 assures that the query graph of the query which generated L is among the candidate graphs after building the lattices. By merging nodes representing identical database tables, the complexity of the individual graphs decreases with every merge step performed, as the number of nodes inside a graph is reduced.

Unless the depth of the tree that the join chains originated from is very low, there will be a number of nodes referencing identical database tables. This

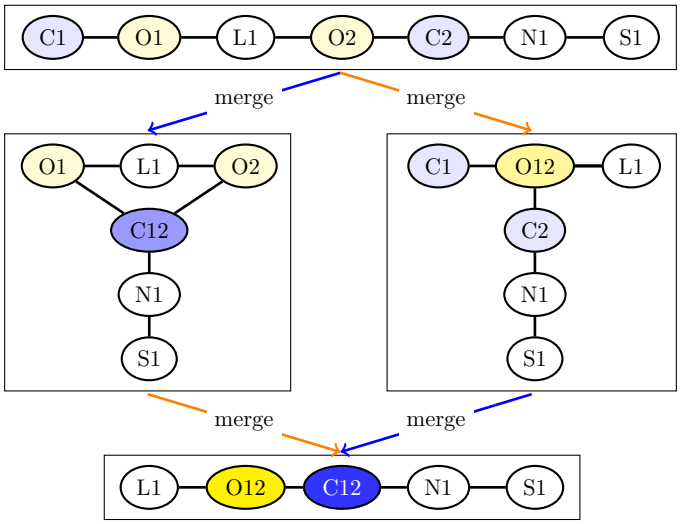


Figure 5.8: Example lattice

leads to multiple merge steps that might be needed to generate a fully merged graph, as seen in Figure 5.8. In this situation it is important to maintain the unique identifiers of merged nodes during intermediary merge steps to be able to distinguish nodes, as they could be mapped to the same identifier after performing pairwise merging.

The process of generating a set of lattices starts with adding the root of the lattice and performs a merge operation for each possible pair of nodes referencing identical database tables, which will result in a row of new graphs. To successfully merge a pair of two nodes, the resulting merged node retains the union of edges and neighbors of the two original nodes. To indicate that the resulting node is a product of a merge step, it will have a modified identifier which consists of the table name and the two number identifiers of the original nodes appended to each other. After this first iteration, the method continues with the graphs resulting from the first merge step and performs further merging to expand the lattice with new graphs until all possible merge steps have been performed and all graphs have been inspected.

When adding new graphs to the lattice, the lattice keeps track of the number of merge steps needed to generate the newly added graph and keeps graphs with the same amount of merge steps in the same set. This allows us to establish the lattice structure with the initial join chain at the top and the fully merged graph with no nodes referencing identical tables at the bottom.

Handling Duplicate Graphs. The two join chains visualized in Figure 5.7(a) are identical with respect to the set of referenced database tables, and both contain two instances of the Customer relation. Figure 5.7(b) shows that both of these join chains collapse to the same chain after performing a merge step on the two Customer nodes. The resulting graph connects the entity can-

didate Customer relation to the ranking candidate Orders, but also retains the join between the Customer and the Nation relation. The join with the unmarked Nation node appears to not have an impact on the result of the query, as it does not connect the entity table to a ranking candidate, but it may still have an influence on the outcome of a query utilizing predicates to filter Customer entities based on attributes of the Nation relation. Thus, it is important to retain the unmarked nodes, even when they are "dangling" nodes as shown in the example.

After merging the nodes in the two join chains of the example, one of the result graphs can be eliminated due to being a duplicate, therefore reducing the overall number of candidates. Keeping track of duplicate graphs is essential to reduce the computation time of the upcoming steps, which require database interactions and therefore represent a bottleneck of the overall computation.

5.3.5 Step 5: Query Building

Converting a query graph into an SQL query requires extracting various pieces of information to form the SELECT, FROM, and WHERE clause. The query graph stores this information and facilitates the construction of SQL queries.

Each join-query gets assigned a cost value that is used for prioritizing the least costly candidate joins to be tested first. We use the estimated size of the join-result table R as an indicator of how expensive is a candidate join. The relative size of R can be calculated by a simple formula as proposed by Swami and Schiefer [SS94].

$$|T_1 \bowtie T_2| = \frac{|T_1| \times |T_2|}{max(d_1, d_2)}$$

where d_1 and d_2 are the cardinality of the join-columns in T_1 and T_2, respectively.

The queries are then sorted in ascending order of their cost and handed to the next step of the computation, the Instance Verification, which aims for eliminating some of the query candidates before the final candidate list is passed on the next components of the framework.

5.3.6 Step 6: Instance Verification

During Instance Verification the scores present in the input list L will be compared to actual values in the database columns to rule out candidate joins which do not contain suitable ranking columns. Since the database interactions needed for this step are expensive, it is important to make use of an efficient verification approach which keeps the number of tuples involved in the database operations to a minimum.

Queries may contain joins with tables which are not connecting the entity table to a table containing candidate ranking columns. The goal of Instance Verification is to eliminate candidate ranking columns which cannot generate the

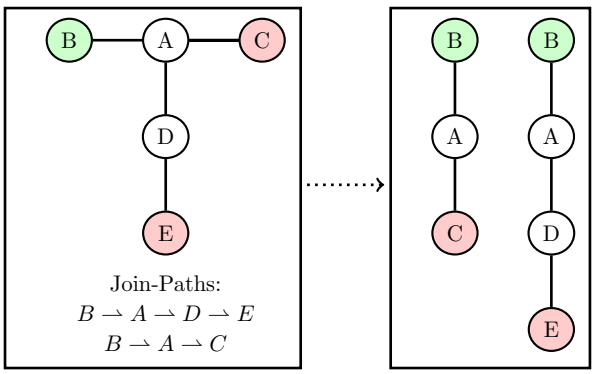


Figure 5.9: Join path representation of different query graphs

input scores. Hence, our goal is to represent each candidate join as one or multiple paths between the table containing the entities and the tables containing the ranking columns.

Figure 5.9 displays candidate joins represented by their query graphs, the entity table is marked in green while the tables containing ranking candidates are marked in red. The first, shown in Figure 5.9(a), can be represented by the two paths $B \rightharpoonup A \rightharpoonup D \rightharpoonup E$ and $B \rightharpoonup A \rightharpoonup C$.

If a query graph can be represented by two or more paths of various lengths, it usually shares those paths with less complex queries that have already been covered by Instance Verification, as the arriving queries are ordered by their estimated cost. Two less complex queries are shown in Figure 5.9(b), each representing one join path of the query in Figure 5.9(a). Reusing results of already processed queries can speed up the verification of more complex queries.

To generate the join paths used by Instance Verification, Dijkstra's Shortest Path Algorithm [CLRS01] is used on the query graphs. Dijkstra's Algorithm will calculate the shortest distance, meaning the least amount of joins needed, from the entity node to each of the ranking nodes, which will result in the respective join paths being built.

The Prefix-path Relationship. One important assumption about the OLAP-queries generating the input list L is that all joins which are not connecting an entity table to a ranking table do not increase the number of tuples used for score aggregation, hence they are used for applying filters. Otherwise, one would aggregate over duplicate values in the ranking column, resulting from joining the ranking table with another table in pk/fk direction.

Having calculated a join path for each candidate join query, we can use them to reduce the computation time of the actual Instance Verification. This can be achieved by keeping track of the validity of previously verified queries. If a valid candidate query's join path is a prefix of another query's join path, that query can also be considered a valid candidate. This is the case because the

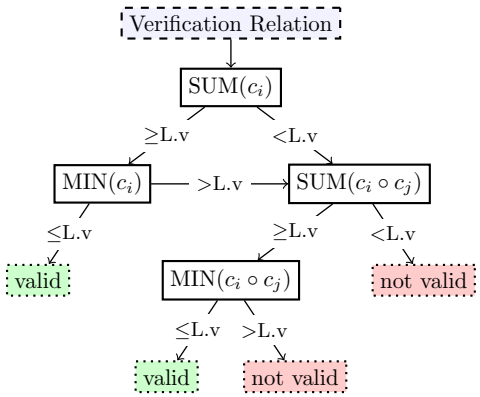


Figure 5.10: Basic classification of candidate queries

less complex query already contains suitable ranking columns, which are also present in the more complex one. This complex query may apply further filtering or not, hence making it a valid candidate query in any case as the boundary conditions for the ranking columns stay the same. In the opposite situation, where a non-valid candidate's join path is a prefix of another query's join path, the algorithm can make use of this result by not testing the already invalidated columns again. For queries which do not share a prefix with an already verified query we can make no assumptions and therefore a full verification needs to be performed.

Candidate Join Query Validation. By default, Instance Verification takes the top entity from the input L and evaluates the previously computed join path on the database by executing the respective SQL query. It is possible to consider more entities from L, resulting in a higher computation time, but also with the possibility of eliminating more ranking candidates.

The result of joining the top entity along the join path is a verification relation V, which contains all possible ranking columns. In order to be a valid ranking candidate, a column has to be able to generate the score in L corresponding to the entity used for generating V. When aggregating the values in a ranking candidate column, all tuples corresponding to the input entity will be used to generate this aggregated score. Therefore this score cannot be directly compared to the score in L, since the aggregation generating the input score may have used only a subset of those tuples, as additional predicates may have filtered some of them out. Hence, score boundaries can only be defined by utilizing the *sum* and *min* aggregation functions on the columns of this table.

Figure 5.10 depicts the decision tree showing how a query will be classified as either a valid or non-valid candidate join. The values of the columns being considered are denoted as c_i and c_j. Note that for a query to be valid candidate, only a single column or column pair needs to be classified as valid, whereas for the query to be non-valid, all columns and column pairs need to be classified as non-valid.

The final list of valid candidate queries is ranked by the calculated cost value and handed to the next component of the framework for discovering possible filter predicates and the aggregation function.

5.4 Optimizations

In this section we propose an advanced classification system to improve the ranking of the candidate join trees, along with other optimizations to decrease the running time of the extended framework.

The ranking of candidate join queries has the most influence on the overall runtime of the algorithm, as each candidate query has to be executed on the database to compare the actual results to the given input list. In case of long-running queries that contain many cost-intensive joins of large tables, each execution of a candidate query is very time consuming. Our goal is therefore to find the correct query with the first candidate joins that are handed to the rest of the framework, reducing the amount of mismatches to a minimum.

5.4.1 Improved Query Ranking

The improved query ranking is achieved by identifying possible aggregation functions and columns during the Instance Verification step and by putting the respective queries into candidate lists with higher priority. During the identification of the specific aggregation functions, one has to be aware of their particular characteristics which may result in varying precision of identifying the individual functions.

Queries using either the *avg* or the *max* aggregation function can be identified more precisely than queries using a *sum* aggregation, since the former two aggregations' output values lie within the range of a column's original values. Furthermore, the *max* aggregation can be identified more reliably than the *avg* aggregation.

After building the priority lists, the candidate join queries inside will be ranked again based on their cost value. The aforementioned lists not only contain the queries which will most likely lead to a matching result query, but also carry information about the aggregation functions which should be considered in the next steps of the framework. To achieve the improved query ranking, more database interactions are needed during the Instance Verification step, which results in a certain overhead being added to every execution. Instead of only considering the sum and the minimum of the values inside a column, now also the mean value and the standard deviation are being computed. This additional information will be used by the algorithm in conjunction with further heuristics to build the priority lists. The improvement in the query ranking outweighs the overhead and we show this in the experiments.

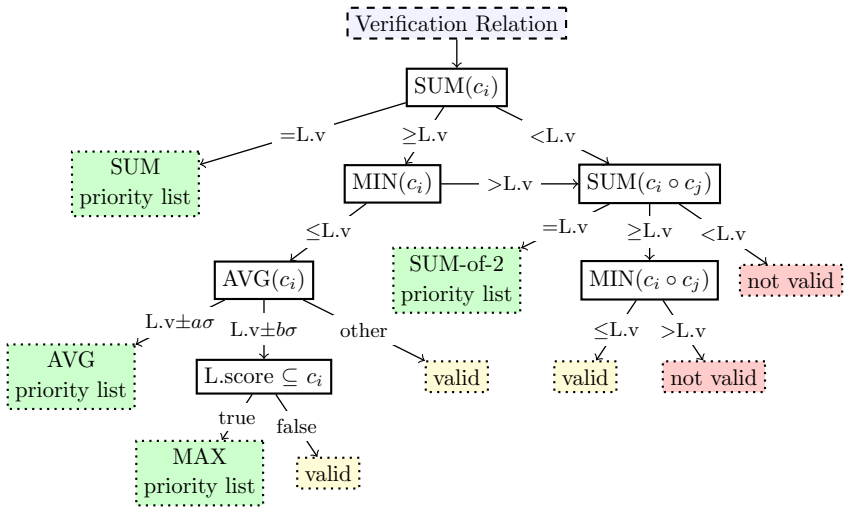


Figure 5.11: Advanced classification of candidate queries

We introduce heuristics that can be applied to build the priority lists, each for the different ranking criteria supported by our system. If the score of an entity lies within a-times the standard deviation of the mean value of the column, then the candidate query containing this column will be entered into a certain priority list. The parameter a should be set sufficiently low, so that few false positives occur, but also high enough to exclude false negatives. Experiments have shown that for the different ranking criteria, a specific margin of the standard deviation fulfills this requirement for most queries.

5.4.2 Advanced Classification of Queries

The generation of priority lists, and therefore the improved query ranking model, makes use of an advanced classification tree, which is illustrated in Figure 5.11. Each of the previously mentioned heuristics for identifying candidate ranking criteria is present inside this classification tree, along with the corresponding priority lists.

A condition containing $L.v$ refers to the score value of the input entity, while $L.score$ refers to a set of score values of the input list L. Note that it is possible to follow multiple edges when checking the conditions in the decision tree, as these are not necessarily mutually exclusive. However, we assume the conditions associated to the edges are checked from left to right and therefore follow a strict order. If the query is not put into any of the various priority lists, it will always be categorized as either *valid* or *not valid* analogous to the baseline Instance Verification approach.

Fast Verification. If a candidate query has been categorized into one of the priority lists, it is very likely to result in a match. Hence the latter parts of the

framework should begin processing this query and all related queries as soon as possible. In order to speed up the remaining Instance Verification process, this optimization called Fast Verification allows to skip through the remaining candidates if at least one query has already been put into a priority list.

During Fast Verification, for the remaining candidates, only the join-paths will be inspected. If a join-path of a query inside the priority list is a sub-path of a path of the currently inspected candidate, this candidate will also be put into the same priority list. Otherwise this candidate join query will not undergo any further inspection and will be added to the list of valid join candidates. Thus, if no matching query was found by processing the candidate joins inside the priority lists, such a join query will be handed to the finding filtering predicates component.

5.5 Discovering Filter Predicates and Ranking Criteria

The techniques for discovering filter predicates and ranking criteria are extensively discussed in Chapter 4. The approach there, considers working on a single input table R, which can now be generated in advance by executing the valid candidate joins. As the result of Instance Verification is a ranked list of join-query candidates, they can be executed in order of their rank to generate the table serving as input for the remaining parts of framework, as shown in Figure 5.4.

To keep the memory usage low, these tables are filtered to only contain the entities from the top-k input list. We denote the joined table that contains only tuples from the input entities as R' and store it in-memory. The system first identifies a set of *candidate predicates* which can be of the form $P_1 \wedge P_2 \ldots \wedge P_m$, where P_i is an atomic equality predicate of the form $A_i = v$ (e.g., country = 'Sweden'). For each candidate predicate there must exist at least one tuple t_i for each entity in L with $P(t_i)$ assessing to true.

A naive approach would take all possible predicates as candidates and proceed to finding ranking criteria. This would significantly reduce performance, since the expansion of the set of candidate predicates would explode. The system utilizes an apriori-style algorithm that uses the anti-monotone property of the criteria what is considered a candidate predicate. It starts by identifying atomic candidate predicates with size $|P| = 1$ and iteratively creates larger conjunctive predicates by efficiently using the ones created in the previous step.

In order to identify suitable ranking criteria, the system uses the set of determined candidate predicates. It utilizes small data samples, histograms, and simple descriptive statistics which are computed upfront, thus selecting suitable columns and aggregation functions as ranking criteria and avoids touching actual table data. Queries are created by combining the selected ranking criteria with the candidate predicates which are efficiently verified on R'. The queries that

Parameter	Value
depth d	5
minJoins	0
maxJoins	3
maxQueries	10

Table 5.1: Parameters used in the experiments

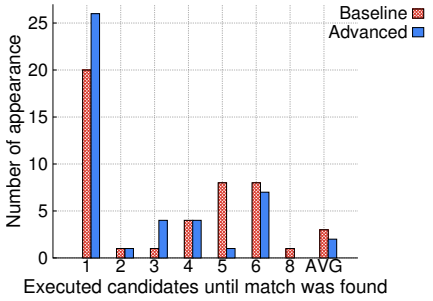


Figure 5.12: Number of executed candidate queries

produce results that match the input list L are marked as candidate queries, which because of the possible presence of false positives (cf., Chapter 4), need to be verified on the base table that contains *all* tuples, not only tuples with the input entities.

In the last component of our framework, the candidate queries are evaluated using the base table data. Unlike the Instance Verification step presented in Section 5.3, where the focus is whether the joined tables *are suitable* to produce the input list, here full-fledged candidate queries are executed and their results are compared if they *match* the input list. The queries that match the input list are returned to the user as valid queries.

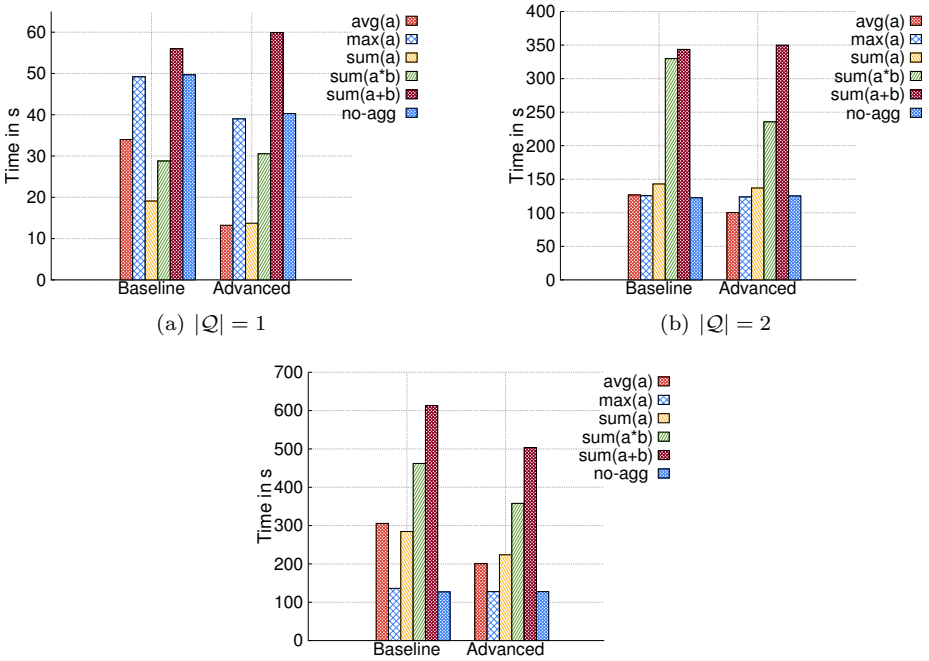
5.6 Experimental Evaluation

The implementation of our framework was done in Java. The experiments have been conducted on a machine running Ubuntu Linux 14.10, powered by two Intel Xeon E5-2603 hexa-core CPU at 1.60GHz, with 128GB of RAM, using Oracle JVM1.8.0_45 as the Java VM (limited to 20GB memory). The tables are stored in a PostgreSQL 9.4 database, with B+ tree indexes created on the possible entity columns.

5.6.1 Dataset and Workload

Dataset. The evaluation of our system was done using the TPC-H [TPC] benchmark. We created a a 10GB dataset, i.e., a scale factor 10 instance of the benchmark.

Workload. The input used for the experiments consists of 46 query outputs. The queries used to generate the outputs make use of varying scoring functions:($max(a)$, $avg(a)$, $sum(a)$, $sum(a+b)$, $sum(a*b)$, and no aggregation), between one and three join operations and between zero and three predicates. Note that, when discussing the different query types, we only write the ranking criterion.



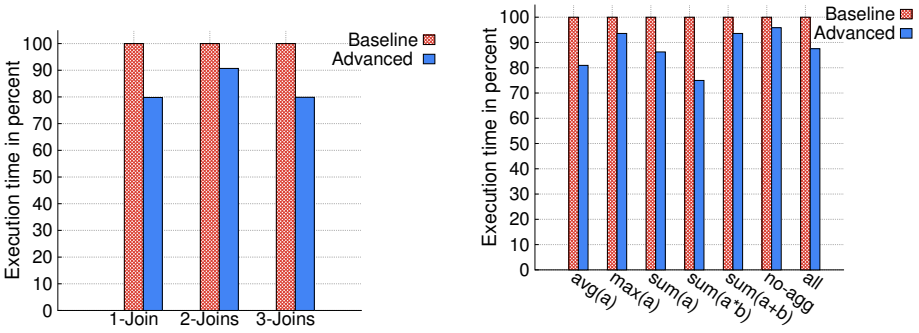
(a) $|\mathcal{Q}| = 1$ (b) $|\mathcal{Q}| = 2$

(c) $|\mathcal{Q}| = 3$

Figure 5.13: Average time to find queries with different query graph size $|\mathcal{Q}|$

All experiments have been conducted with the parameter values shown in Table 5.1. The depth d for generating the tree in Step 2 of the computation has been set to 5, queries with zero to three joins will be considered as candidates and the maximum amount of candidates that will be considered for finding filtering predicates and ranking has been set to 10. These values can be chosen based on the database schema complexity and the estimated complexity of the query to be reverse engineered and allow to limit the execution time of the algorithm by not generating an unnecessary large amount of candidates. Tuning parameter a has been set to reasonably small values to allow proper categorization into the priority lists. There were three queries that were outliers, executing their candidate queries on the database took significantly more time than any of the remaining queries in the workload. In order not to skew the results, we do not include these queries in the presented results.

If the basic framework without any optimizations was used to run the experiments, we refer to it as the *baseline approach*, otherwise, we refer to the *advanced approach* when all of the optimizations were utilized. Note that in the evaluation we focus on identifying the join conditions of the query. Finding filtering predicates and ranking criteria is discussed in depth in our previous work. Furthermore, the reported results focus on efficiency of discovering the first valid query in the results that are presented.



(a) By number of joins (b) By scoring function

Figure 5.14: Relative execution times

5.6.2 Results

Finding a Matching Query. First, we want to point out that all of the 46 queries have been successfully reverse engineered by our framework, regardless of whether the baseline approach or the advanced approach was applied.

The goal of the optimizations of the advanced approach is to improve the ranking of the candidate join queries, which are given as input to the next steps of the framework. The optimal ranking has the query which is most likely to be a match at the top of the candidate list. Therefore, we show the quality of the ranking by inspecting how many candidate joins have to be executed until a matching query is found.

Figure 5.12 shows this statistic for the two presented approaches. On average, the advanced approach inspects approximately two queries to find a match while the baseline approach needs about three inspections. The baseline has many cases of inspecting five queries before finding a match and even has a single case where eight inspections were needed. Depending on the time needed to inspect a single candidate query, the difference between inspecting three or two candidate queries on average may be significant, as the following experiments show.

The process of finding a matching query consists of the six steps to generate the candidate list and further finding filtering predicates and ranking criteria by the framework until a matching query is discovered. Figure 5.13 displays the average time until a matching query is found, grouped by the different scoring functions supported by our framework and the number of joins.

Figure 5.14(a) compares the relative execution times of the advanced approach to the average running time of the baseline as the reference point. On average the advanced approach is faster in reverse engineering a query regardless of the amount of joins and it performs 10–20% faster than the baseline approach.

Figure 5.14(b) not only groups the results by scoring function instead of the

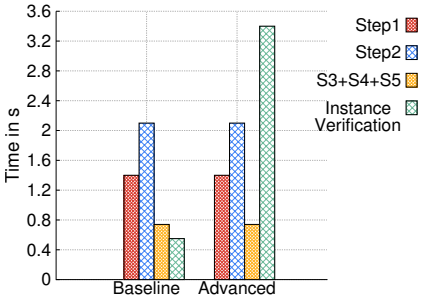


Figure 5.15: Execution times of the join-framework steps

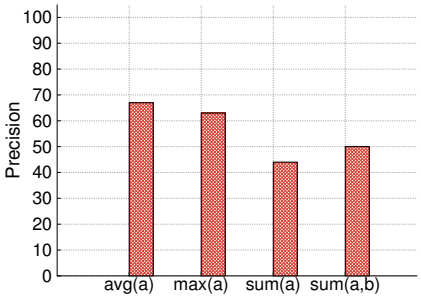


Figure 5.16: Precision of categorizing the result query into the correct priority list

number of joins, but also averages over the **relative** execution times of reverse engineering individual queries. This is done to make the results between the queries with a different amount of join operators comparable, since they heavily vary in the absolute time needed to find a match. Without this adjustment, the results of reverse engineering queries with three joins would dominate the overall result.

As can be seen in Figure 5.14(b), reverse engineering queries with the *max* aggregation function or no aggregation function profits the least from the optimizations of the advanced approach, with only about 5% decrease in running time averaged over the amount of joins. The $sum(a, b)$ queries profit the most and are discovered about 30% faster. Overall, queries are found almost 14% faster on average when using the advanced approach, ignoring the amount of joins and the used scoring function.

Generating the Candidate List. To gain insight into the execution time of the six steps of finding the join predicates, Figure 5.15 shows the median execution times of these steps based on the 46 test queries.

Steps 1–5 do not vary between the baseline and the advanced approach and also remain relatively constant between the individual queries. Among the first five steps, Step 1 and 2 take the most time, as they are interacting with the database to gain information about the schema. The added up execution times of Steps 3–5 range in the milliseconds even though they are dealing with the more complex data structures, which shows the impact of even the smallest database interactions. The Instance Verification step differentiates the baseline from the advanced approach and the experiments show a significantly increased execution time in the latter, due to the extended classification process. However, this is a small cost to pay compared to the benefit that is gained in the next steps. Looking only at the six steps to generate the list of candidate join queries, we can say that the candidate joins are found within a few seconds for both the baseline and the advanced approach.

Precision of the Advanced Approach. The improved ranking of the advanced approach can only be achieved if the candidate query which resembles the query generating the input list L is categorized into a priority list for early inspection. To gain insight into the accuracy of the advanced classification process, we can check how often candidate queries are put into the priority lists which correspond to the matching queries' scoring function.

Figure 5.16 shows the precision of this classification grouped by the different priority lists. When comparing this statistic to the one showed in Figure 5.14(b), one can see that the accuracy of the classification does not correlate with the actual performance gain of making use of the improved ranking. This is the case because each aggregation function has a different impact on the quality of the candidate ranking of the baseline approach. The *max* aggregation seems to result in a good ranking of query candidates when using the regular Instance Verification, while the advanced Instance Verification step invests a lot of time into checking columns for the inclusion of several input scores to put a candidate into a MAX priority list, resulting in a large overhead.

Overall 50% of the scoring functions involving two columns and an average of almost 70% of the scoring functions involving a single column have been correctly identified during Instance Verification of the advanced approach.

5.6.3 Lessons Learned

The main findings from the above experimental results are:

- Both the baseline and the advanced approach were able to reverse engineer every single query of the 46 input queries.
- The advanced approach, which makes use of the advanced classification system, executed fewer candidates before it found the matching query.
- The advanced classification system also comes with an overhead. However, this overhead pays off, since the performance gain from improving the ranking of the candidate list depends on the time needed to execute the individual candidates, which are expensive because of the present joins.
- The advanced approach always outperforms the baseline and more complex join queries have bigger performance gain. Having more join operations leads to slower candidate query execution, thus the ranking played a bigger role when reverse engineering these complex queries.
- The advanced approach reduced the time needed for reverse engineering join queries by up to 20% compared to the baseline approach.

5.7 Summary

In this chapter, we presented a framework for reverse engineering top-k database join queries. This is a challenging problem, given the small input and the amount

of database tables that have to be considered as potential join partners due to the lack of information to filter candidates out. Having only entities and aggregated scores to find and filter out candidates, we had to find a way to interact with the database system to efficiently classify the large amount of candidate queries. This was achieved with the proposed Instance Verification step, which was refined with the advanced classification system to significantly improve the query ranking. The introduction of priority lists improves the ranking of candidate join queries, which ensures that less false positives will be considered by the remaining parts of the framework.

Chapter 6

Concept and Computation of Ranking-based Dominance

6.1 Introduction

The emerging field of data exploration aspires at developing approaches that allow users to easily access vast amounts of available data, in order to learn hidden insights and answering their information needs. In the previous chapters of this thesis we described methods that help users in unearthing knowledge by learning interesting properties of their input lists via the different constraints of the reverse engineered queries. Moreover, previous research in data exploration includes topics like assisting users in discovering relevant data objects [DP13, DPD14] and refining their interests through faceted search [KHP10, KJTN14]. In this chapter, we address the problem of exploring data through dominance between ranked entities. It is based on our own publication at ExploreDB 2018 [PM18].

Consider a user Bob who wants to buy a car. Bob is not a car enthusiast and does not know too much about cars yet, but there are a couple of models that he likes, say a BMW 118i and an Audi A3. Now, he would like to learn about other cars that outperform his favorites in some or all rankings where they appear together, i.e., the car models that are *dominating* his input. Considering the rankings in Figure 6.1, Bob can learn that the Mercedes A220 is a model that outperforms the two input cars across all four rankings. Thus, he learns that this model is always better than his favorites and that he should probably consider it as a potential alternative for buying. Moreover, he can examine what are the constraints and ranking criteria where the Mercedes is better, for instance, in cars of type = Hatchback and ranked by power. He also wants to see which car models are *dominated* by these two and he gets the Citroen, Ford, and Chrysler

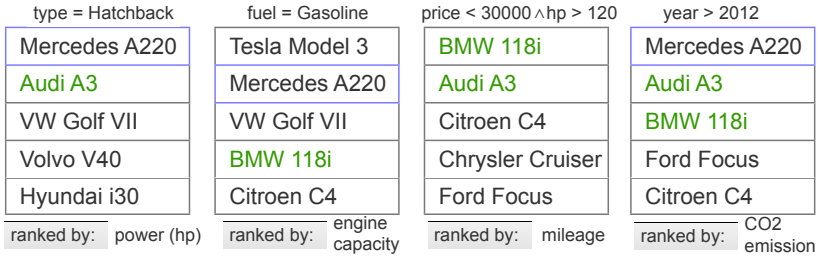


Figure 6.1: Example of dominance between ranked entities

models as being dominated by the input. Since his favorites are always ranked better than these three models, he can disregard them as potential models of interest.

In this work, we investigate the dominance relationship between ranked entities and develop algorithms to explore data summarized in rankings by computing dominant entities. We formulate the dominance relationship of entities in a set of top-k rankings and focus on the different nuances that this relationship entails. The database of rankings could be given by any platform that generates or crowdsources rankings based on objective measures or human assessments, such as `ranker.com`, interpreting Web tables as rankings, or generated by transforming facts of knowledge bases (KBs) into rankings [IMS13].

6.1.1 Problem Statement

We consider a database T of rankings τ_i. Each ranking has a domain D_{τ_i} of items it contains. None of the rankings contains any duplicate items and there are no ties between items. The rank of an entity e in a ranking τ is given as $\tau(e)$. In this work, we assume that $\tau(e)$ takes values from 0 to $k-1$ (instead of 1 to k).

Given a set of input entities I, the objective in this work is to efficiently and effectively determine all entities e that dominate all entities in I, i.e., are ranked higher in all lists where they co-occur. More formally, we are looking for the set of entities:

$$\hat{D}(I) = \{e \mid \forall e_i \in I \ : (\exists\, \tau \ : \ (e, e_i) \in \tau) \wedge \\ (\forall \tau \ : \ (e, e_i) \in \tau \Rightarrow \ \tau(e) < \tau(e_i))\}$$

We are also interested in finding the set of dominated entities $\check{D}(I)$ that are ranked lower than $e_i \in I$ in all lists τ they appear together, i.e., where $\tau(e_i) < \tau(e)$.

The problem is extended by relaxing the criterion of the entities dominating the input from across all common lists to a certain portion of the lists, as the original criterion can often lead to empty results, particularly for large input sets I. Thus, given a user-specified threshold θ, dominating entities are to be considered those that are ranked higher than the input in more than θ of their

common rankings. We elaborate on what are the consequences and how we adjust our methods by having this relaxed criterion. We define the dominance relationship more formally in Section 6.2.

To keep the presentation concise, in the remainder of the chapter we will consider only finding the entities that are dominating I; however, all methods can be applied for also finding the dominated entities.

The problem of finding dominating entities interactively given ad-hoc user inputs, requires efficient computation and scans over potentially a very large number of rankings. Hence, we consider suitable index structures and develop algorithms that make use of early pruning, compare them to baseline methods, and discuss on how the pruning reduces data access and affects the efficiency of computation.

6.1.2 Contributions and Outline

With this work we make the following contributions:

- We present COMPETE, an efficient, effective, and extensible approach for exploring data through ranked lists that computes dominating and dominated entities.
- We define the entity dominance relationship over a set of rankings and consider different types of dominance given a set of input entities, referred to as group dominance.
- We introduce a model for estimating the number of dominating entities for a set of input entities, which is utilized for auto tuning which type of dominance is looked for.
- We derive algorithms for finding dominating entities using different performance optimizations, in particular, harnessing the early-stopping potential when computing the top-m most dominant entities.
- We report on the results on an extensive experimental evaluation using data from the *Internet Movie Database* (IMDb) and a synthetically generated dataset.

The remainder of the chapter is organized as follows: Section 6.2 presents the approach and the nuances of the entity dominance relationship. Section 6.3 introduces a model for estimating result size and auto tuning method execution. Section 6.4 describes the baseline algorithms and first optimizations, while Section 6.5 introduces methods for efficiently computing the top-m most dominant results. Section 6.6 reports on the results of an experimental evaluation. Section 6.7 presents a summary of the chapter.

τ_1	τ_2	τ_3	τ_4	τ_5
e_1	e_3	e_2	e_1	e_4
e_4	e_7	e_6	e_2	e_2
e_2	e_1	e_3	e_3	e_5
e_6	e_6	e_5	e_6	e_6
e_8	e_8	e_4	e_4	e_3

Figure 6.2: Sample set of rankings

6.2 The COMPETE Approach

COMPETE comprises various ways to determine dominance, suitable index structures, and algorithms. Let us start by describing the concept of dominance of entities according to an input set of entities.

6.2.1 A Model for Dominance Across Rankings

To do so, it first needs to be stated how the notion of dominance known from skyline literature translates to our setup. This is in fact straightforward. Recall that we consider a database of rankings to be given and that for two entities being present in the same ranking, we assume that it is more preferable to be placed in a position at the top of the ranking, rather than in the tail. One can then say that the first entity dominates the latter, in this list. For multiple lists, we can then simply define that **an entity e_a dominates an entity e_b**, denoted as $e_a \prec e_b$ in a set of rankings T *iff* for all rankings $\tau_i \in T$ that contain both e_a and e_b, $\tau_i(e_a) < \tau_i(e_b)$. In words, the dominated entity is always ranked worse than the dominating one. As stated earlier in the problem description, we assume that the rankings do not have ties[1].

Consider the sample set of rankings presented in Figure 6.2. The entity e_1 is said to dominate e_2, since in their common rankings, τ_1 and τ_4, e_1 is ranked higher than e_2: $\tau_1(e_1) < \tau_1(e_2)$ and $\tau_4(e_1) < \tau_4(e_2)$. Similarly, e_2 dominates e_3. An important observation is that the dominance relation is not transitive, due to ranking incompleteness [KML08]. Thus, when comparing e_1 and e_3 we see that neither one dominates the other, since $\tau_2(e_3) < \tau_2(e_1)$ and $\tau_4(e_1) < \tau_4(e_3)$. The inherent incompleteness of top-k rankings implies that two entities may not co-occur in any of the lists. For instance, e_5 does not co-occur with e_7 and e_8 in any of the rankings shown in Figure 6.2.

[1] If rankings were allowed to have ties (cf., [FKM+04]), this definition would need to be slightly adapted, such that the dominating entity is at least in one ranking ranked better and in others not worse (in fact, this is the standard definition of dominance in skyline research, where dimensions of interest are numeric or at least comparable). This slight change would, however, be of only marginal influence to the overall approach and is not discussed further for brevity.

6.2.2 Fractional Dominance

Definition 12 Significance of Dominance
The significance of dominance $sig(e_a, e_b)$ of an entity e_a over another entity e_b is defined as the ratio between the number of rankings where e_a dominates e_b and the number of rankings in which they co-occur:

$$sig(e_a, e_b) = \frac{|\{\tau_i \mid \tau_i(e_a) < \tau_i(e_b)\}|}{|\{\tau_i \mid (e_a, e_b) \in \tau_i\}|}$$

We refer to $|\{\tau_i \mid \tau_i(e_a) < \tau_i(e_b)\}|$ as the **degree of dominance *deg*** and call $|\{\tau_i \mid (e_a, e_b) \in \tau_i\}|$ the **support of dominance $\boldsymbol{\sigma}$** between two entities.

The significance of dominance allows for obtaining insight on how two entities relate to each other in terms of high or low placements in the rankings. The largest significance of dominance amounts to 1 and happens when e_a is ranked higher than e_b (or vice-versa) in all lists where they co-occur. Note that, $sig(e_a, e_b) = 1 - sig(e_b, e_a)$. Using the significance of dominance, we define:

Definition 13 Fractional Dominance
We say an entity e_a fractionally dominates an entity e_b given a threshold θ, denoted as $e_a \prec^{\theta} e_b$ iff $sig(e_a, e_b) > \theta$.

The definition for fractional dominance allows for some leniency in computing dominating entities, however, θ should still be chosen reasonably high. If an entity e_a is ranked higher than e_b in 95 out of 100 rankings where they co-occur together, and having $\theta = 0.9$, we say $e_a \prec^{\theta} e_b$, since $sig(e_a, e_b) = 0.95 > \theta$.

6.2.3 Group Dominance

The notion of dominance is now extended to dominance of an entity over a group of entities.

Definition 14 Group Dominance
An entity e dominates a group/set of entities I, denoted as $e \prec_g I$ iff e dominates all $e_i \in I$, i.e., $\forall e_i \in I : e \prec e_i$.

This intuitive definition is very restrictive for two reasons. First, an entity needs to dominate all entities in the group and, second, dominance between two entities is only defined if the two entities co-occur in at least one ranking. This means that in particular for larger inputs I, the result might be rather often empty, as with larger input I, it gets more and more unlikely that an entity is co-occurring with *all* elements of I. To ameliorate this harsh behavior of group dominance, we can relax it in a way such that entities in I that do not overlap with an entity e are simply ignored to assess whether or not $e \prec_g I$.

Definition 15 Partial Group Dominance
An entity e partially dominates a set of entities I, denoted as $e \prec_p I$ iff (i) e co-occurs with at least one entity $e_i \in I$ and (ii) e dominates all entities $e_i \in I$ that it co-occurs with.

If an entity e dominates the set I, then it also partially dominates I, i.e., it holds that $e \prec_g I \Rightarrow e \prec_p I$, but not the other way around. A dominating entity needs to co-occur with each and every one of the entities in the input set and is dominating all of them. However, a partially dominating entity does not need to co-occur with all the entities in I, but it dominates the entities from I with which it co-occurs in the rankings.

Let us consider the input set $I = \{e_5, e_6\}$ and the sample rankings in Figure 6.2. We find that the entity e_2 dominates I, since it dominates both e_5 and e_6. On the other hand, e_1, e_2, and e_7 partially dominate I, considering that e_1 and e_7 dominate e_6 and $\sigma(e_1, e_5) = \sigma(e_7, e_5) = 0$. Note that if $\sigma(e_7, e_5) > 0$, but e_7 is ranked lower than e_5 in some ranking list (thus is not dominating e_5), then e_7 would not be part of the partially dominating entities, even though it dominates e_6. Partial dominance is important when the input contains multiple entities. For instance, if $|I| = 5$ and an entity e is dominating 4 of the 5 entities in the input, then e would probably still be an interesting result to the user, however will not qualify as a result with group dominance.

Overall, if we replace the dominance relationship $\prec$ in the two definitions above by fractional dominance $\prec^\theta$, we obtain four different combinations of group dominance:

- **Group Dominance** $\prec_g$
- **Partial Group Dominance** $\prec_p$
- **Fractional Group Dominance** $\prec_g^\theta$
- **Fractional Partial Group Dominance** $\prec_p^\theta$

6.2.4 Assessing Dominance

The resulting set of dominating entities for a given input set I can be very large, especially for the partial group dominance. In order not to overwhelm users, in such cases, the resulting entities need to be presented in a sorted manner, by decreasing importance. The notion of fractional group dominance already tells that some results are better than others—an entity that is ranked higher in more rankings is yielding a higher significance of dominance. Thus, the significance of dominance can be used as a scoring function upon which the dominating entities are ranked. Given an input set I and a dominating entity e the significance of dominance of e over the input set I can be computed as:

$$sig(e, I) = \sum_{e_i \in I} sig(e, e_i)$$

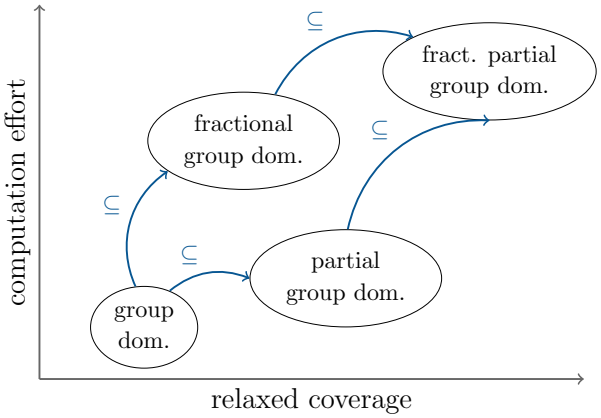


Figure 6.3: Correlation of the different types of dominance

With non-fractional dominance $sig(e, e_i) = 1$, hence, there is a need of quantifying the dominance differently. Following the definition of significance of dominance, we consider using the degree of dominance as a scoring function. Intuitively, the results that have the largest number of common rankings with the input can be considered as most related to the input and provide the most information to the user. Thus, we compute the degree of dominance of e over the input set I as:

$$deg(e, I) = \sum_{e_i \in I} deg(e, e_i)$$

with the most dominating entities having the largest degree of dominance $deg(e, I)$. In case of ties, we suggest breaking them by the global number of rankings an entity is contained in (i.e., its popularity). Furthermore, since the degree of dominance $deg(e, I)$ is a monotone ranking function, it is suitable for using top-k-style processing techniques with early stopping [FLN01]. In Section 6.5 we present algorithms that emphasize on computing the top-m most dominating entities.

6.3 Auto Tuning

Figure 6.3 depicts the containment relations between the four considered types of dominance and puts them into perspective regarding computational effort and strictness of the dominance concept. The group dominating entities need to dominate all entities from the input I, thus making group dominance the strictest. Moreover, they are the most interesting for the user, since they are in relation with all the input entities. Group dominance is also the most efficient to compute, as explained in Sections 6.4 and 6.5. Partial group dominance is less strict than group dominance, with partial group dominating entities dominating at least one of the input entities I. It requires also more computational effort than group dominance. Furthermore, the set of group dominating entities is a

subset of the ones computed with partial group dominance, since $e \prec_g I \Rightarrow e \prec_p I$. The remaining correlations can be seen in Figure 6.3; we focus on computing results with non-fractional dominance.

A naive approach is to look for group dominance and if there are no results returned, subsequently automatically look for partial group dominance. However, this approach is inefficient—for some input sets of entities, both methods for group and partial group dominance need to be executed. On the other hand, the most efficient approach would always execute the method for group dominance when it would result in a non-empty result set and otherwise execute directly the method for partial group dominance, without "wasting" the runtime for group dominance at all. In order to do this, knowledge about the size of the result set is necessary. Thus, given a set of input entities I, we need to estimate the size of the result set $|\hat{D}(I)|$ containing the entities that dominate those in I. We propose an approach that estimates $|\hat{D}(I)|$ by estimating the number of dominating entities for pairs of input entities in I. Given a pair of entities (e_a, e_b), with sets of dominating entities $\hat{D}(e_a)$ and $\hat{D}(e_b)$, respectively, we can compute $\hat{D}(e_a, e_b) = \hat{D}(e_a) \cap \hat{D}(e_b)$ as the set of entities that dominate both e_a and e_b, i.e., $\hat{D}(e_a, e_b) \prec_g (e_a, e_b)$. The size of $|\hat{D}(e_a) \cap \hat{D}(e_b)| \leq min\{|\hat{D}(e_a)|, |\hat{D}(e_b)|\}$, with equality in the case $\hat{D}(e_a) \subseteq \hat{D}(e_b)$ or vice-versa. Thus, in order to estimate the size of $\hat{D}(e_a, e_b)$, inspired by the Jaccard index [Jac12], we introduce an estimation coefficient ec:

$$ec = \frac{|\hat{D}(e_a) \cap \hat{D}(e_b)|}{min\{|\hat{D}(e_a)|, |\hat{D}(e_b)|\}}$$

where $0 \leq ec \leq 1$ and which can then be used to compute $|\hat{D}(e_a) \cap \hat{D}(e_b)|$. The higher the value of ec, the more elements from the smaller set of dominating entities $\hat{D}(e_a)$ or $\hat{D}(e_b)$ are dominating both e_a and e_b; smaller ec signifies fewer entities dominating both e_a and e_b.

We compute the average estimation coefficient $avg(ec)$ for a dataset of rankings T by using a set of sample entities S from the dataset T. For each pair of entities $(e_a, e_b) \in S$, the number of dominating entities and their estimation coefficient is computed. Then, $avg(ec)$ for T is the average over ec for each entity pair in the sample. We can now use $avg(ec)$ to estimate the size of group dominating entities $|\hat{D}(I)|$ for a given set of input entities I:

$$est(|\hat{D}(I)|) = min\{|\hat{D}(e_i)|, e_i \in I\} \times avg(ci)^{|I|-1}$$

Note that in addition to $avg(ec)$, in order to estimate the size of $\hat{D}(I)$, we also need the number of dominating entities for each entity in the dataset T. This, as well as calculating $avg(ec)$, is a one-time computation and can be done in a pre-processing step for the entire dataset T.

Having the estimation of the size of $|\hat{D}(I)|$ allows for a smarter approach in looking for dominating entities for a set of input entities I, as shown in Algorithm 10. Now, depending on the estimated $est(|\hat{D}(I)|)$ and the wanted m most dominating entities, the method for group dominance or the one for partial

```
method: smartDominance
1  compute est(|D̂(I)|)
2  if est(|D̂(I)|) ≥ m
3     findDominance⋆(I, m)
4     if |D̂(I)| < m
5        findPartialDominance⋆(I, m)
6  else
7     findPartialDominance⋆(I, m)
```

Algorithm 10: Computing the top-m most dominant entities with smart dominance

group dominance can be executed accordingly. Note that if the actual number of returned results with group dominance is insufficient, the approach still falls back to running the method for partial group dominance, thus ensuring that the user gets a non-empty result set in cases when $est(|\hat{D}(I)|)$ is incorrect.

6.4 Algorithms

After the definition of different flavors of dominance and how to quantify the utility of a discovered dominance relationship in terms of significance and relatedness, we now discuss ways to efficiently compute dominating entities for a user-provided input set I. Note that for reasons of brevity we only show the details of finding the dominating entities; analogous steps are taken for computing the dominated entities.

6.4.1 Basics

Let us start with the example illustrated in Figure 6.4 that underpins the general concept of utilizing the contents of rankings to determine dominance. Assume that the entities VW and Fiat are the input I and that they occur in the rankings τ_{10}, τ_{12}, and τ_{18}. In the simplest case of trying to compute group dominance, the rankings in which VW occurs are falling into two partitions, each, the entities that are dominated by VW and the ones that dominate VW, indicated by green and red background in the illustration. By definition of group dominance, entities in the 'red' parts of the rankings are ruled out. From the 'green' parts, only Audi fulfills dominance, while GMC only satisfies the partial dominance (because it is simply not present in τ_{12} and τ_{18} and hence does not co-occur with Fiat).

Consequently, the first algorithm we can think of inspects for each input entity the ranking lists that contain it, determines the aforementioned partitions of each list and returns those entities that are in the 'green' area and never

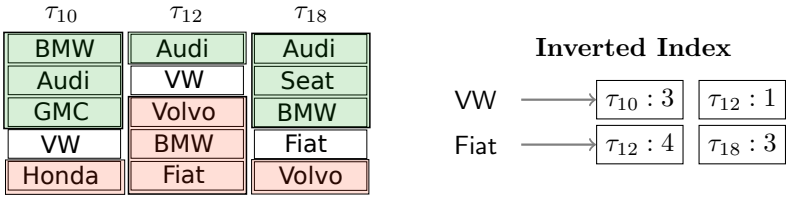


Figure 6.4: Illustration of finding dominant entities

method: **findDominanceBaseline**

```
1  for each e_i ∈ I
2    rList_{e_i} := I.getPostingList(e_i)
3    for each τ ∈ rList_{e_i}
5      add all e s.t. τ(e) < τ(e_i) to G_{e_i}
6      add all e s.t. τ(e) > τ(e_i) to R_{e_i}
7    D̂_{e_i} = G_{e_i} \ R_{e_i}
8  D̂ = ⋂_{e_i ∈ I} D̂_{e_i}
9  return D̂
```

Algorithm 11: Finding group dominance

in any 'red' partition. The group dominating entities are those that are in the intersection of the ones that individually dominate the input entities. The corresponding pseudocode is depicted in Algorithm 11. The list of rankings that contain VW is retrieved from the inverted index and entities ranked higher (the 'green' parts) are added to a set $G_{\mathsf{VW}} = \{\mathsf{BMW}, \mathsf{Audi}, \mathsf{GMC}\}$ and the ones ranked lower (the 'red' parts) to a set $R_{\mathsf{VW}} = \{\mathsf{Honda}, \mathsf{BMW}, \ldots\}$ (Lines 5–6 in Algorithm 11). The entities that occur in G_{VW}, but not in R_{VW}, dominate VW: $\hat{D}_{\mathsf{VW}} = \{\mathsf{Audi}, \mathsf{GMC}\}$. Similarly, $\hat{D}_{\mathsf{Fiat}} = \{\mathsf{Audi}, \mathsf{BMW}, \mathsf{Seat}\}$ and the final result with group dominance is computed as $\hat{D} = \hat{D}_{\mathsf{VW}} \cap \hat{D}_{\mathsf{Fiat}} = \{\mathsf{Audi}\}$.

6.4.2 Pruning, Blacklisting, and Result Sharing

The reader has perhaps noticed that Algorithm 11 is wasteful for two reasons. First and foremost, entities that are already discarded as not dominating for a certain input entity e_i, are still considered as candidates for the remaining entities in I. Second, the sets of dominating entities $\hat{D}_{e_i}$ are computed individually for each input entity e_i and these are later intersected to get the final result, even though it is necessary an entity to dominate all $e_i \in I$. Following the example presented in Figure 6.4, when looking for the dominating entities for Fiat, BMW would still be processed even though it is already established that it does not dominate VW. Similarly, Seat will also be considered as a potential dominating entity, but it is not among the entities that dominate VW and therefore cannot dominate I. This unnecessary processing diminishes the efficiency of the system. Computation can be shared across all input entities

method: **findDominance+**

```
1  order e_i ∈ I in ascending order of σ(e_i)
2  /* start with rarest e_i and compute dominance */
3  L_{e_i} := I.getPostingList(e_i)
4  for each τ ∈ L_{e_i}
5    for each e ∈ τ
6      if blacklist.contains(e) then skip e
7      if τ(e_i) < τ(e) then
8        if D̂.contains(e) then prune and blacklist e
9        if ¬Ď.contains(e) then add e to Ď
10     /* same is done for τ(e) < τ(e_i) in case of Ď */
11 for each remaining e_i in ascending order of σ(e_i)
12   L_{e_i} := I.getPostingList(e_i)
13   for each τ ∈ L_{e_i}
14     if D̂ = ∅ return there are no results
15     for each e ∈ τ
16      skip e if not already a result
17      if τ(e_i) < τ(e) ∧ D̂.contains(e) then
18       prune e from D̂
19 return D̂
```

Algorithm 12: Optimized group dominance

in I, thus reducing the data access and computation runtime. We propose two optimization techniques that improve the efficiency: **prune-and-blacklist** and **result-sharing**.

Algorithm 12 incorporates these two optimizations for computing group dominance. The prune-and-blacklist strategy (Lines 4–10 in Algorithm 12) eliminates entities early and avoids their re-processing by placing them in a blacklist. Consider the entity BMW in our example, which after processing τ_{10} and τ_{12}, is observed as being ranked both higher and lower than VW. Hence, BMW is not a dominating entity and it can be pruned from the results and consequently blacklisted. Now, let us consider there are more rankings in the example dataset. In the remainder of the processing, BMW can be skipped, both when processing the rankings with VW and those with Fiat, the ranking comparison to the input entities avoided, thus, reducing the data-access footprint of the algorithm. Note that checking if an entity is blacklisted does incur some overhead, however, this cost is much smaller than processing these entities that cannot anyhow be final results.

With group dominance, additional data access can be saved from sharing the intermediate results computed for the individual input entities in previous steps, by using the dominating entities computed for one input entity e_i and considering only those in each subsequent step if they also dominate the remaining input entities in I, similarly to *Term-at-a-Time* (TaaT) query processing [MRS08] in

search engines. Moreover, the input entities are processed in ascending order of their support, starting with the rarest entity e_i which occurs in the lowest number of rankings, since a small initial result set is desirable.

The result-sharing optimization is shown in lines 11–19 of Algorithm 12. Consider again the example shown in Figure 6.4. After processing VW and computing $\hat{D} = \{\mathsf{Audi}, \mathsf{GMC}\}$, the algorithm needs to consider only $\{\mathsf{Audi}, \mathsf{GMC}\}$ as entities that can have group dominance over I. In this way, during the processing of the rankings with Fiat, all entities that are not in $\hat{D}$ are skipped, regardless if they are already blacklisted or not. For instance, processing Seat is unnecessary, since it does not dominate VW, even though this is due to the fact that $\sigma(\mathsf{VW}, \mathsf{Seat}) = 0$, rather then Seat being blacklisted. Significant data processing is saved by considering only the entities $\hat{D}$. Note that $\hat{D}$ will get smaller with the processing of each additional $e_i \in I$, meaning more entities will be skipped in the latter stages of the algorithm. Moreover, $\hat{D}$ could become empty, leading to early termination of the algorithm (Line 14 in Algorithm 12).

6.4.3 Computing Partial Dominance

An entity e that partially dominates the input set I, dominates some of the entities $e_i \in I$ and does not have any common rankings with the remaining entities $e_j \in I, i \neq j$, i.e., $\sigma(e, e_j) = 0$. Thus, computing partial group dominance differs to group dominance in two key aspects. First, the result-sharing technique is not possible, since a partially dominating entity e does not need to dominate all $e_i \in I$. Therefore, the individual result sets $\hat{D}_{e_i}$ for each $e_i \in I$ need to be computed separately and the final result is the union of the entities in each $\hat{D}_{e_i}$. Second, there might be **false positive** results in $\hat{D}$ and these need to be removed.

Consider again the example in Figure 6.4, with $I = \{\mathsf{VW}, \mathsf{Fiat}\}$. However, let us now consider that the algorithm processes the rankings containing Fiat first. We get $\hat{D}_{\mathsf{Fiat}} = \{\mathsf{Audi}, \mathsf{BMW}, \mathsf{Seat}\}$ and Volvo in the blacklist. Then, after processing VW, $\hat{D}_{\mathsf{VW}} = \{\mathsf{Audi}, \mathsf{GMC}\}$, but this time BMW is also in the blacklist, since it was seen as a lower-ranked entity (in the red area) than VW in τ_{12}. Thus, BMW cannot partially dominate I, even though it dominates Fiat. Partially group dominating entities are those that are in the union of the individual $\hat{D}_{e_i}$ filtered of false positives: $\hat{D} = \bigcup_{e_i \in I} \hat{D}_{e_i} \setminus$ blacklist.

Partial dominance is less restrictive, thus, yielding more results than group dominance. Moreover, the inability to use the result-sharing strategy makes partial dominance more expensive to compute than group dominance. Nevertheless, the prune-and-blacklist strategy still manages to avoid significant computational overhead.

6.4.4 Computing Fractional Dominance

Finding fractional dominance allows for some leniency in computing dominating entities and a fractionally dominating entity needs to be ranked higher than an entity in more than θ fraction of their common lists. In **fractional group dominance**, an entity needs to have common rankings with and dominate all entities from the input in more than a θ fraction of the lists. The approach is similar to the one presented in Algorithm 12, however, there is a difference in when an entity is pruned and blacklisted.

Consider again the example dataset in Figure 6.4, with $I = \{\mathsf{VW}, \mathsf{Fiat}\}$, and $\theta = 0.6$. In Section 6.4.1, we showed that $\mathsf{Audi} \prec_g I$. Let us consider adding the following ranking to the example dataset:

$$\tau_7 = [\mathsf{VW}, \mathsf{BMW}, \mathsf{Audi}, \mathsf{Honda}, \mathsf{Fiat}]$$

In τ_7, VW is the top-ranked entity, making the rest of the ranking, including Audi, a dominated 'red area' partition, which would mean that Audi is not group dominating I anymore. Moreover, Algorithm 12 would blacklist Audi after processing the rankings τ_7 and τ_{10}, since it occurs as both ranked higher and lower than VW. However, fractional group dominance is not as strict as group dominance and Audi needs to be ranked higher than VW in at least $\theta \times \sigma(\mathsf{VW}, \mathsf{Audi}) = 1.8$ rankings, or ranked lower in at most $(1-\theta) \times \sigma(\mathsf{VW}, \mathsf{Audi}) = 1.2$ rankings. When computing fractional dominance, Audi would be blacklisted and pruned once there are 1.2 (thus, in fact, 2) or more rankings found where Audi is ranked lower than VW.

Thus, given an input set of entities I, an entity $e \prec_g^\theta I$ iff for each $e_i \in I$:

$$|\{\tau_i \mid \tau_i(e_i) < \tau_i(e)\}| \leq t_\theta(e, e_i)$$

where the threshold number of rankings t_θ is computed as $t_\theta(e, e_i) = (1 - \theta) \times \sigma(e, e_i)$. Note that t_θ is different for each pair of entities (e, e_i).

Going back to the example, in order to compute the threshold number of rankings $t_\theta(\mathsf{VW}, \mathsf{Audi})$, the support $\sigma(\mathsf{VW}, \mathsf{Audi})$ needs to be known. We compute the support of all possible entity pairs in the dataset in a pre-processing step. Furthermore, additional bookkeeping of the number of times $\tau(\mathsf{VW}) < \tau(\mathsf{Audi})$ (and vice-versa) is necessary. Since Audi does not reach the threshold t_θ for both VW and Fiat, it is computed as a fractionally dominating entity.

Fractional partial group dominance requires the same criterion as partial group dominance regarding the number of input entities dominated, hence similar steps for avoiding false positives need to be taken. Similarly, it cannot benefit from the result-sharing technique. The aforementioned threshold t_θ is used in computing this flavor of dominance and an entity e is pruned and blacklisted once found as ranked lower in more than $t_\theta(e_i, e)$ number of rankings, for any $e_i \in I$.

Figure 6.5: Inverted index providing co-occurrences information.

6.5 Emphasizing on Top-m Results

It is easy to see that the aforementioned algorithms compute the exhaustive, correct result. However, this could lead to a potentially very large number of results, which brings the need to assess the dominance of an entity, as presented in Section 6.2.4. The benefits of quantifying the dominance of an entity e by assigning it a dominance score value $score(e)$ are twofold: (1) it facilitates result presentation to the user by ranking the dominating entities according to $score(e)$; (2) the result space can be pruned by stopping early in the processing, once the top-m most dominant results are computed.

We use the degree of dominance $deg(e, I)$ as the scoring function $score(e)$ that quantifies the dominance of e over the input set I. Intuitively, the results that co-occur the most with the input entities are most closely related to the user interest. With non-fractional dominance, $deg(e, e_i) = \sigma(e, e_i)$. Since $\sigma(e, e_i)$ is pre-computed and known for each entity pair in the dataset, it is used as the scoring function that is aggregated.

Computing the dominating entities can be done by inverted-index-only access. We can intersect and process the posting lists of each input entity $e_i \in I$ with the posting lists of all entities e in the dataset that co-occur with any of the input entities, i.e., have $\sigma(e, e_i) > 0$. Following, we present algorithms that, given an input set of entities I, compute the top-m results.

6.5.1 Top-m Group Dominance

The method for computing the top-m most dominating entities with group dominance is described in Algorithm 13. For each of the input entities $e_i \in I$, score ordered posting lists are created with the support of the co-occurring entities as the score. Thus, an entity e_i will have all its co-occurring entities in the posting list, ordered by their support in descending order. Consider the sample set of rankings shown in Figure 6.4 and the input set $I = \{\mathsf{VW}, \mathsf{Fiat}\}$ as a running example in explaining the algorithms. The posting lists containing the co-occurring entities for the input entities are shown in Figure 6.5. The posting list for VW, for instance, contains all the entities that co-occur with VW in the dataset, together with their support, i.e., the number of rankings in which they appear together. We can see that Audi occurs together with VW in two rankings, while with Honda in one ranking. The cost of creating these posting lists is marginal compared to the rest of the computation and we create them dynamically for the given input entities, but this can also be done in a preprocessing step for the entire dataset.

method: **findDominance⋆**

```
1  create the co-occurrence posting lists for each e_i ∈ I
2  scan posting lists (round-robin)
3  consider e = ce_id(i) in posting list for e_i
4  high(i) = cscore(i)
5  /* compute score(e) */
6  if e ∉ top-m then
7     look up score(e_j, e) for all j ≠ i
8     skip e if score(e, e_j) not available for some j
9     score(e) = sum{score(e, e_j) | j = 1 ... |I|}
10 /* checkDominance(e) */
11 for each  e_i ∈ I and e
12    open the posting lists L_{e_i} and L_e for processing
13 while at least one L not entirely processed
14    align the posting list L_e with each of L_{e_i}
      on matching ranking τ
15    if L_{e_i}.τ(e_i) < L_e.τ(e)
16       return no dominance for e
17 if e is dominating /* update top-m */
18    if score(e) > min-m then
19       add e to top-m and remove min-score e'
20       min-m = min{score(e') | e' ∈ top-m}
21 /* update upper bound from current scan line */
22 ub = sum{high(i) | i = 1 ... |I|}
23 if ub ≤ min-m then return top-m
```

Algorithm 13: Computing the top-m most dominant entities with group dominance

Keep in mind that there are now **two types of posting lists:**

- for each $e \in T$ with postings $(\tau : \tau(e))$ containing information in which rankings an entity e occurs and its position therein
- for each input entity $e_i \in I$ with postings $(e : \sigma(e, e_i))$ containing information on which entities e co-occur with e_i in the dataset and what is their support

Algorithm 13 is adopting Fagin's Threshold Algorithm (TA) [FLN01] in the way the co-occurring entities e of the input are processed, how their score is computed, and in the way the top-m entities are maintained (Lines 2–9, 18–23 in Algorithm 13). However, an entity e can only enter the top-m results if it is, in fact, a dominating entity over I (Line 17 in Algorithm 13). Thus, ***e* is checked for dominance** over I by retrieving and processing posting lists from the positional inverted index (Lines 11–16 in Algorithm 13).

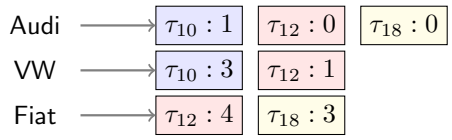


Figure 6.6: Processing of posting lists

The co-occurrence posting lists are scanned in round-robin fashion and the score of an encountered entity e therein is computed by looking-up the score in the posting lists for each input entity e_i. Since in order for $e \prec_g I$ it must co-occur with all $e_i \in I$, there is no need of its further processing if it does not appear in some posting list, say for an input e_j (Line 8 in Algorithm 13). This check is important as it averts the expensive dominance check in the next step, for entities that anyway cannot be dominant. In processing Audi it is confirmed that it co-occurs with each input entity (with $score(\mathsf{Audi}) = 4$), hence its dominance over I needs to be checked. Therefore, the posting lists of Audi, VW, and Fiat are retrieved from the positional inverted index (shown in Figure 6.6) in order to check the position of Audi with regard to the position of each entity in I. In order to make this alignment efficient, the posting lists in the inverted index are kept ordered in ascending order of the ranking identifier τ_{id}. Then, the processing of the posting lists is done similarly as in Document-at-a-Time (DaaT) query processing [MRS08]. The posting list of Audi is aligned interchangeably on a common ranking with each input entity. This is done separately with each input entity, since Audi does not need to co-occur with all of them in a single ranking. Furthermore, in order to increase efficiency, skipping with galloping search [MRS08] is being used when traversing the posting list and looking for a matching ranking identifier.

Once a common ranking is found (marked with different colors in Figure 6.6), the position of Audi is checked against the input entity. For instance, $\tau_{10}(\mathsf{Audi}) < \tau_{10}(\mathsf{VW})$, thus the processing can continue and the same will be done for τ_{12} and τ_{18}. Note that in this example, all postings are processed, since there are very few rankings in the dataset and they are a common ranking for some pair of the entities discussed. When examining the dominance of an entity e, the investigation is stopped if $L_{e_i}.\tau(e_i) < L_e.\tau(e)$; e is then disregarded as a result and the stopping avoids further accesses to the posting lists L.

However, Audi is always ranked higher than VW and Fiat, thus $\mathsf{Audi} \prec_g I$ and it is added to the top-m results. The upper bound of the maximum score that an entity can reach is computed by aggregating the score from the current scan line and this upper bound is used to check whether or not a not-yet seen entity can enter the top-m results [FLN01]. The upper bound $ub = 4$ and say we are computing the top-1 result. Since $score(\mathsf{Audi}) = ub = 4$, no other entity in the dataset can have a higher score and Audi is the top-1 result. Alternatively, if the upper bound score is larger than the entity with the smallest score in the top-m, the algorithm continues with the next entity in the co-occurrence posting

method: **checkFractionalDominance**

```
1  for each  e_i ∈ I and e
2    open the posting lists L_{e_i} and L_e for processing
3    /* compute the t_θ for each e_i */
4    t_θ(e, e_i) = (1 − θ) × σ(e, e_i)
5    rankedLower(e, e_i) = 0
6  while at least one L not entirely processed
7    align the posting list L_e with each of L_{e_i}
     on matching ranking τ
8    if L_{e_i}.τ(e_i) < L_e.τ(e)
9      rankedLower(e, e_i)++
10   if rankedLower(e, e_i) > t_θ(e, e_i)
11     return no dominance for e
12 return e is dominating
```

Algorithm 14: Method for checking fractional dominance

lists of the input entities. Note that similar steps are taken for computing the entities that are dominated by I and a separate top-m list is maintained.

6.5.2 Top-m Partial Group Dominance

The algorithm for computing partial group dominance is very similar to the one for finding group dominance presented in Algorithm 13. Recall that an entity $e \prec_p I$, if it dominates some of the entities $e_i \in I$ and does not have any common rankings with the remaining entities $e_j \in I$, i.e., $\sigma(e, e_j) = 0$. Thus, when computing partial group dominance, the condition in Line 8 in Algorithm 13 is dropped. Hence, GMC will be processed and checked for dominance, even though it does not co-occur with Fiat in any of the rankings in the dataset. The remainder of the algorithm is the same as when computing group dominance and the check for dominance of an entity remains unchanged. Using this approach avoids computing false positives, which is the case with the methods presented in Section 6.4. By using a positional inverted index, the dominance of an entity e is checked for all input entities $e_i \in I$ at once, by processing the inverted index posting lists.

The number of entities e when computing $\prec_p I$ is always higher than when computing $\prec_g I$. Furthermore, computing partial group dominance is more expensive than computing group dominance. Consider our example, with partial group dominance each entity found in the co-occurrence posting lists in Figure 6.5 needs to be processed, while with group dominance, GMC, Honda and Seat can be skipped.

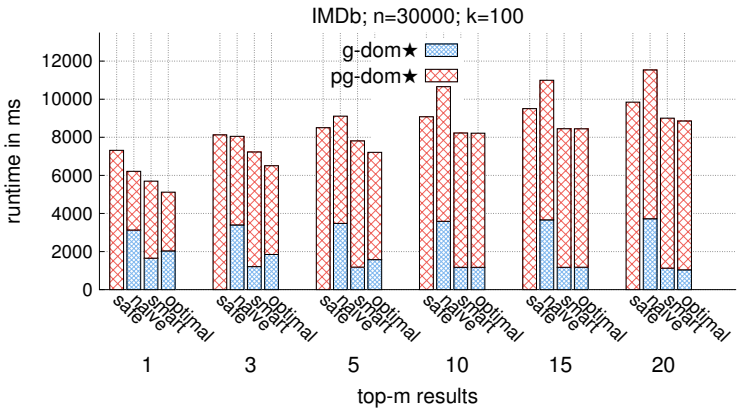


Figure 6.7: Total runtime of the entire workload with varying m

6.5.3 Top-m Fractional Dominance

An entity $e \prec^{\theta} I$ iff it is ranked higher than $e_i \in I$ in more than θ fraction of their common rankings. Thus, e can still be a fractionally dominating entity even if it is ranked lower than an input entity e_i in multiple rankings and the threshold number of rankings $t_\theta(e, e_i)$ depends on the input threshold θ, as shown in Section 6.4.4. The process of computing the top-m fractionally group-dominant entities is similar to finding the top-m group dominant entities presented in Algorithm 13, with the **difference in the checking for dominance**. The check for fractional dominance is shown in Algorithm 14.

First, for each of the input entities e_i, the threshold number of rankings $t_\theta(e_i, e)$ where e can be ranked lower (or higher when looking for dominated entities) is computed and their posting lists are retrieved from the positional inverted index. Furthermore, for each pair (e, e_i) counters for maintaining how many times e is ranked lower than each e_i are initialized. Then, the posting lists are traversed in a Document-at-a-Time (DAAT) fashion [MRS08] with the same nuances as described for computing group dominance. Once a common ranking is found for e and an input entity e_i, the positions from the postings are compared and if $L_{e_i}.\tau(e_i) < L_e.\tau(e)$, the counter rankedLower is increased. The processing of e is stopped if rankedLower$(e, e_i) > t_\theta(e, e_i)$, since e cannot be a fractionally dominating entity. On the other hand, if for all input entities $e_i \in I$, the threshold $t_\theta(e, e_i)$ is not breached, e is returned as being a dominating entity. Note that when looking for dominated entities, a counter rankedHigher(e, e_i) is also maintained and checked against $t_\theta(e, e_i)$. Computing fractional dominance results in processing more postings than with non-fractional dominance.

6.6 Experimental Evaluation

Experiments were conducted on a 2× Intel Xeon 6-core machine, with 128GB RAM, running Ubuntu 14.04 as an operating system and using Java 1.8 (lim-

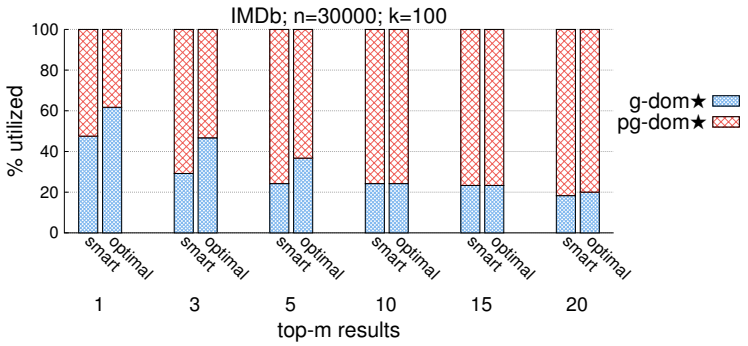


Figure 6.8: Percentage of total workload a method is utilized with varying m

ited to 8GB memory). On system start-up, the dataset, index structures, and statistics are loaded into main memory, thus we examine the performance of all methods as in-memory approaches.

6.6.1 Datasets and Workload

Datasets. We evaluate our approach of computing entity dominance for a given group of input entities using data from two datasets: the IMDb dataset (`www.imdb.com`) and a synthetically generated dataset (SYNT) that was created with the purpose of evaluating the scalability of the presented algorithms. The **IMDb dataset** contains 30000 rankings with top-k size 100 (using the LIMIT clause) and was created by executing (meaningful) queries with different constraints in the WHERE clause and using a variety of aggregation functions as ranking criteria. There are 101 distinct atomic predicates which were combined to create predicates of size 1 to 5, while 13 different ranking criteria were used. For instance, year = 2015, genre = 'Drama', and max(rating) are sample predicates and ranking criterion. These 30000 rankings contain 8850 distinct entities, with a median entity occurrence being 147.5 rankings. The **SYNT dataset** contains 200000 rankings with top-k size 100 created by permuting the rankings in the IMDb dataset, with each ranking being permuted 5 to 10 times. In order to examine the effects of the dataset size n and top-k rankings size, by choosing subsets of the created ranking lists, we created experiments with $n \in \{10000, 20000, 30000\}$ for IMDb, $n \in \{100000, 150000, 200000\}$ for SYNT, and with top-k $\in \{10, 20, 30, 50, 100\}$.

Workload. As workload, we created sets of input entities by using popular actors and actresses. In order to examine the result size and performance of our methods in regard to the number of input entities, we varied the number of the entities in a set I. Thus, we created 30 sets containing 5 entities each and removed entities from them to create sets with smaller size, resulting in total of 150 input sets containing 1 to 5 entities. For each method, we perform the experiments three times for each input set I and report on median performance.

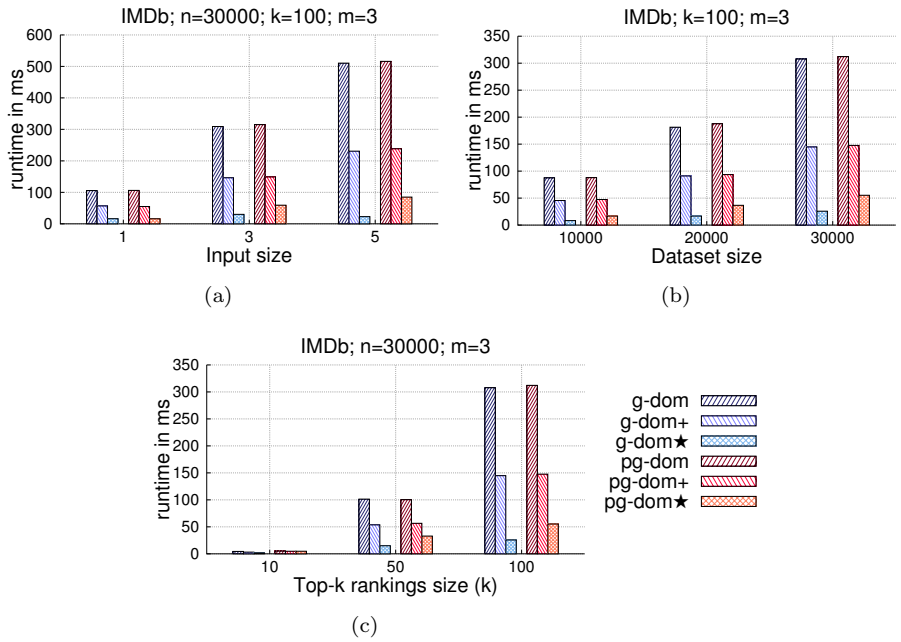


Figure 6.9: Runtime of the methods on IMDb dataset with different (a) input, (b) dataset and (c) top-k size

6.6.2 Competitors and Measures of Interest

We focus on presenting the benefits from the auto tuning model which picks the most promising approach to satisfy a user request, presented in Section 6.3. We label the combined approaches as follows: **smart** is utilizing the probabilistic model in estimating the result sizes and accordingly chooses which method to run, **safe** is always executing partial group dominance as the safe choice in returning results, **naive** is running partial group dominance if group dominance produces empty results, and **optimal** is always running the appropriate method, chosen correctly depending on the input set. Then we look into more detail in measuring the performance (**wallclock time and number of data accesses**) and distinction in results for computing the different combinations of dominance described in Section 6.2.3.

We use the approach presented in Section 6.4.1 as baseline methods which we label as follows: **g-dom** for group dominance $\prec_g$, **pg-dom** for partial group $\prec_p$, **fg-dom** for fractional group $\prec_g^\theta$, and **fpg-dom** for fractional partial group dominance $\prec_p^\theta$. Furthermore, we show the efficiency benefits from the prune-and-blacklist techniques and the algorithms computing the top-m results presented in Section 6.5, while using the exhaustive approach presented in Section 6.4.1, adjusted for each flavor of dominance, as baseline. For the notation, we add a plus sign '+' at the end of each method employing **prune-and-blacklist**, for instance **g-dom+**, and '$\star$' at the end of those focused on the **top-m results**, for example **g-dom$\star$**.

6.6.3 Results

Combined Approach Results

We study the efficiency and benefits of the auto tuning model by comparing it to alternative approaches. Figure 6.7 shows the runtime of the entire workload when executed on the IMDb dataset, with different number of top-m results requested. Furthermore, it shows what part of the total runtime was spent on looking for group dominance (g-dom$\star$) and partial group dominance (pg-dom$\star$), respectively. For example, note that the safe approach always executes pg-dom$\star$, thus no runtime is spent on looking for group dominance. The smart approach falls back to executing pg-dom$\star$ in case the estimate was wrong and g-dom$\star$ did not satisfy the request. Moreover, note that we ignore the input sets in the workload containing only a single entity; there is no difference between group dominance and partial group dominance in such cases. We use the methods focusing on top-m results, since these are the most efficient methods as shown below in the detailed efficiency results.

We observe that the smart approach outperforms both the safe and naive approaches, for all choices of top-m results requested. The estimated result size with the smart approach leads to executing the appropriate method for dominance, thus avoiding the execution of both g-dom$\star$ and pg-dom$\star$ for a single input set, which often occurs with the naive approach. Furthermore, it benefits from running the more efficient g-dom$\star$, where it is plausible, thus outperforming the safe approach. For instance, with $m = 1$, the total runtime of the workload with smart is reduced by 23% when compared to safe, while it improves efficiency by 12% with $m = 15$. Similarly, smart outperforms naive by 10% with $m = 1$ and by 24% with $m = 15$. Moreover, the total runtime of the workload increases with larger m, since more data needs to be processed in order to get more results, thus the execution is stopped at a later stage. We observe that the total runtime of smart is higher than the one of optimal, however the difference is insignificant with larger m. Thus, with $m = 1$ smart is 10% slower than optimal, while, for instance, with $m \geq 10$ the difference in runtime is less than 2%. The reason is that with larger m, the result size is more correctly being estimated by the probabilistic model, resulting in smart choosing the correct method for getting the required m results.

Figure 6.8 shows the percentage of the number of times g-dom$\star$ and pg-dom$\star$ were utilized in the total workload, for the smart and optimal approaches. We observe that with larger m smart estimates the result size more correctly and appropriately chooses the correct method to execute. For instance, with $m = 1$ smart utilizes g-dom$\star$ in 47% of the workload, while the optimal approach results in 61.7%. On the other hand, for example with $m = 10$, smart and optimal both utilize g-dom$\star$ in 24.2% of the workload. Furthermore, we observe that the percentage of utilizing g-dom$\star$ drops with larger m. Group dominance is very strict, thus making it difficult having larger number of results which dominate all the input entities.

Detailed Efficiency Results

We study the efficiency of the different methods and the proposed optimizations for the various dominance relationships in more detail. Note that the results below show the average execution time per input set. Figures 6.9 and 6.10 show the average execution time per input set for each method with different parameters, on the IMDb and SYNT dataset, respectively. Note that Figure 6.10 for the SYNT dataset is in logarithmic scale.

Figure 6.9(a) depicts the performance with different input size and we observe that each of the prune-and-blacklist methods significantly outperforms its baseline equivalent, while the top-m approach improves the efficiency even further. As expected, the runtime increases with larger input size, however this is much smaller with the optimized methods. For instance, the average execution time for g-dom with input size 1 and 5, amounts to 105.5 and 510.5 milliseconds, with g-dom+ equals to 57 and 230.8 ms, respectively, while g-dom$\star$ outperforms both with average runtime of 16.2 and 22.8 ms. The increase in runtime from input size 1 to 3 is by a factor of 2.9 with g-dom, 2.6 with g-dom+, and 1.9 with g-dom$\star$. Moreover, with input size 5, g-dom$\star$ improves upon g-dom by a factor of 22.4, and is faster than g-dom+ by a factor of 10.1.

Figure 6.10(a) shows the runtime by input size with the SYNT dataset. We observe that the gain of the pruning strategies is immense with the larger SYNT dataset. Thus, with input size 5, g-dom+ outperforms g-dom by a factor of 3.2, while g-dom$\star$ is even better, being two orders of magnitude faster than both g-dom and g-dom+. The top-m approach benefits from reduced data access by skipping entities that are not present in the co-posting lists of all input entities and the early stopping once the top-m results are computed. We observe a similar increase in runtime with SYNT as with IMDb when increasing input size for both g-dom and g-dom+. However, interestingly, the runtime with g-dom$\star$ decreases with larger input size. Having a larger dataset introduces more variations in the rankings, with entities being ranked in different ranking positions, hence are found not dominating early in processing their posting lists, thus reducing data access and runtime.

We examine the execution time of our methods with different dataset size. Figures 6.9(b) and 6.10(b) depict the average execution time when varying the number of rankings with IMDb and SYNT, respectively. We observe that, as expected, the execution time increases with more number of rankings in a dataset. With the IMDb dataset, the increase in runtime from using 10k to 30k rankings, amounts to a factor of 3.5 for g-dom, 3.2 for g-dom+, and a factor of 3.1 for g-dom$\star$. However, g-dom$\star$ is significantly outperforming both methods. For instance, with 30k rankings, g-dom$\star$ completes computation in 25.8 ms on average, while g-dom+ and g-dom require 144.8 and 307.9 ms, respectively. The benefits of the top-m approach are even more significant with the SYNT dataset, where the increase in average execution time from 100k to 200k rankings amounts to a factor of 2.1 for g-dom and a factor of 2 for g-dom+. Interestingly, with g-dom$\star$ the runtime increases from 37 to 40 ms, or an increase of only 8%. This is

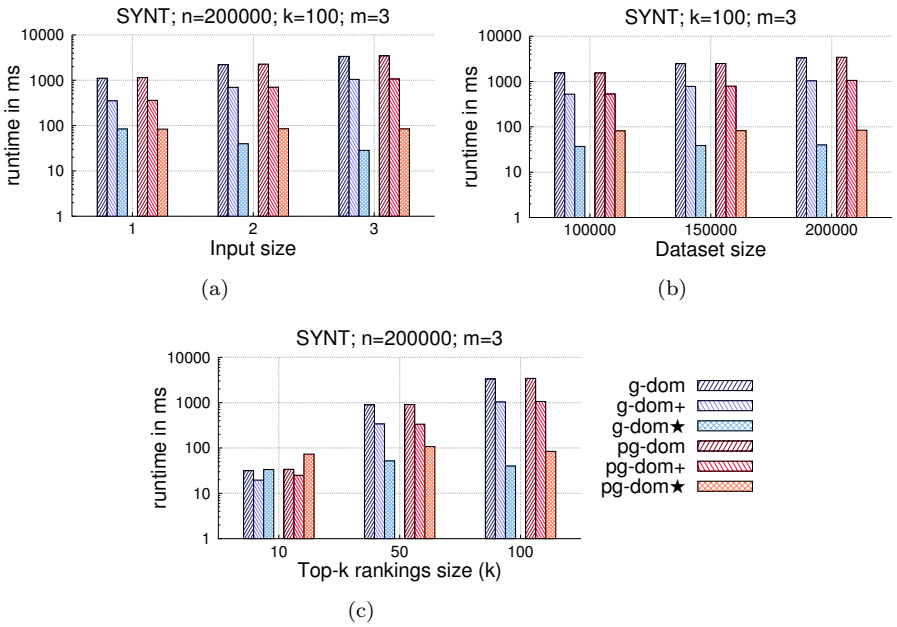


Figure 6.10: Runtime of the methods on SYNT dataset with different (a) input, (b) dataset, and (c) top-k size

due to the nature of the top-m approach, where the variations in a larger set of rankings allows for early pruning of entities that cannot be dominating, as explained before. Moreover, with the largest dataset containing 200k rankings, g-dom$\star$ on average performs by an order of magnitude better than g-dom+, and by two orders of magnitude better than g-dom.

The performance of our methods when varying the size of the top-k rankings is shown in Figures 6.9(c) and 6.10(c), for IMDb and SYNT, respectively. We observe that having larger top-k rankings leads to slower computation time. For instance, with the IMDb dataset, the average execution time with the baseline g-dom is increased by a factor of 68.4 when increasing the rankings size from $k = 10$ to $k = 100$. Similarly, the performance of g-dom+ is reduced by a factor of 45.3, while g-dom$\star$ is performing slower by a factor of 13.6. Having a smaller k means that the input entities might not be present in many rankings in the dataset, thus reducing the number of rankings that need to be processed. Furthermore, small k would mean less number of entities are co-occurring with the input which reduces the possible result space.

We observe similar performance with the SYNT dataset, however, there, the average runtime of the top-m approach is increased mildly with the increase of k, much similar with the behavior shown when varying the dataset size. Larger k means more rankings in the dataset will contain the input entities, where they can be present in different positions in the rankings, which in turn reduces the possible number of results and enables early pruning of entities that are not dominating the input.

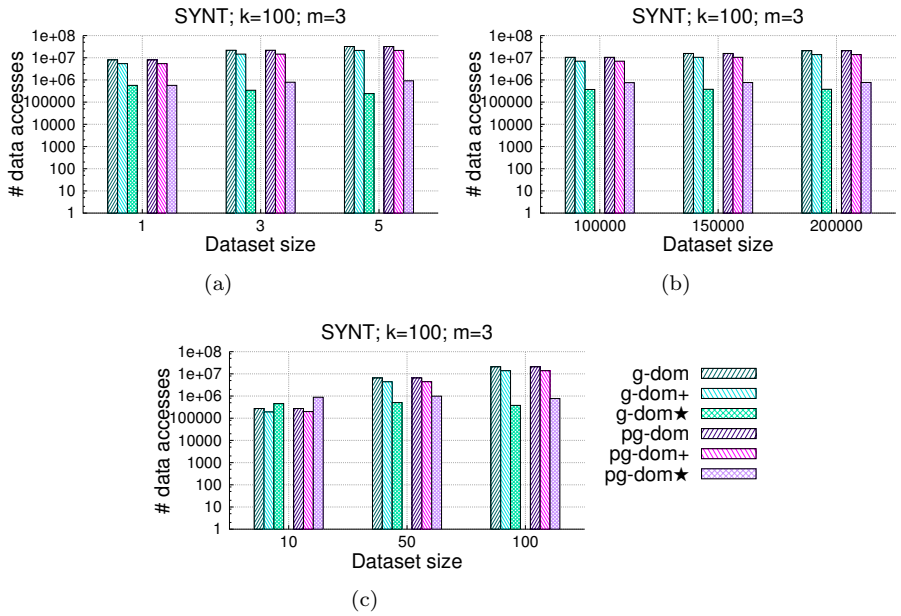
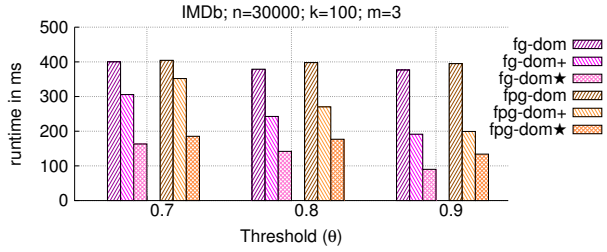


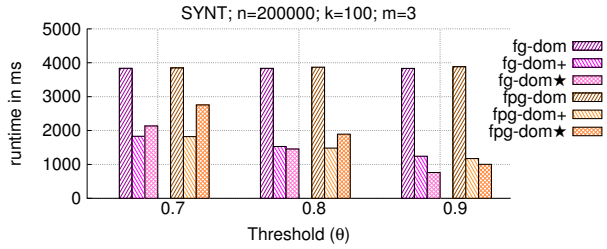
Figure 6.11: Data accesses of the methods on SYNT dataset with different (a) input, (b) dataset, and (c) top-k size

Furthermore, with both datasets, the largest gain of using the top-m approach is with top-k rankings with 100 entities. Thus, on the IMDb dataset, g-dom$\star$ is on average faster than g-dom+ by a factor of 5.6 and is faster than g-dom by a factor of 11.9. The prune-and-blacklist algorithm outperforms the baseline by a factor of 2.1. The gains are even more significant on the SYNT dataset and there, as shown in Figure 6.10(c), g-dom$\star$ is two orders of magnitude faster than g-dom. Similarly, it improves performance by a factor of 25.9 compared to g-dom+, while the prune-and-blacklist method is faster than the baseline by a factor of 3.2. We observe that with the SYNT dataset and $k = 10$, the prune-and-blacklist method is on average faster than the top-m method. The data accesses made with the prune-and-blacklist method are in proportion of nk, i.e., the number of rankings in the dataset n and their size k. On the other hand, the accesses made with top-m strategy depend on the number of distinct entities in the dataset l and the size of the posting list of each entity in the inverted index. Thus, in the results with $k = 10$, there are on average 9023.6 rankings that contain the input, which means there are 90236 entities that are accessed with the prune-and-blacklist strategy. However, there are, on average, 5112.2 distinct entities found in the rankings of which 1337.4 are processed, having an average posting list length of 327.1 postings, thus resulting in 450758.4 accesses. Note that entities and postings are accessed multiple times and additional lookups are necessary to check if an entity is in the blacklist, but the top-m method needs more data accesses.

The prune-and-blacklist and top-m methods prune the result space, thus re-



(a) IMDb



(b) SYNT

Figure 6.12: Comparing the runtime with different threshold θ

ducing the number of data accesses necessary, which leads to faster performance. The number of data accesses, shown in Figure 6.11 for the SYNT dataset, describes a similar behavior among the approaches as for the runtime, i.e., the reduced runtime of g-dom$\star$ over g-dom and g-dom+ closely corresponds to the savings in data accesses made with the top-m approach.

Similar observations are made with the methods for computing partial group dominance. Increasing the dataset and size of the top-k rankings leads to longer execution times with all methods, but the increase is less significant with the top-m method, since it prunes the result space as explained before. Moreover, there are no false positives possible with this method, thus reducing execution times even further. Larger input size leads to larger runtime for pg-dom and pg-dom+. However the runtime of pg-dom$\star$ is decreasing with more input entities. This is due to the nature of group partial dominance, where a dominating entity needs to dominate some of the entities in I, while not co-occurring with the rest. Thus, there is a greater chance of computing the top-m dominating entities faster, since there are more input entities that can be individually dominated and fewer entities from the co-occurrence posting lists need to be processed. This reduces the number of data access, which leads to faster execution times.

The top-m approach pg-dom$\star$ significantly outperforms its two counterparts and the performance difference is similar to the one observed with group dominance. We observe that computing group partial dominance is more expensive than computing group dominance. This is caused by the larger number of entities that potentially can be part of the result with partial dominance. Moreover, partial group dominance does not benefit from the result-sharing optimization, hence pg-dom+ is consistently slower than g-dom+. Furthermore, the top-m

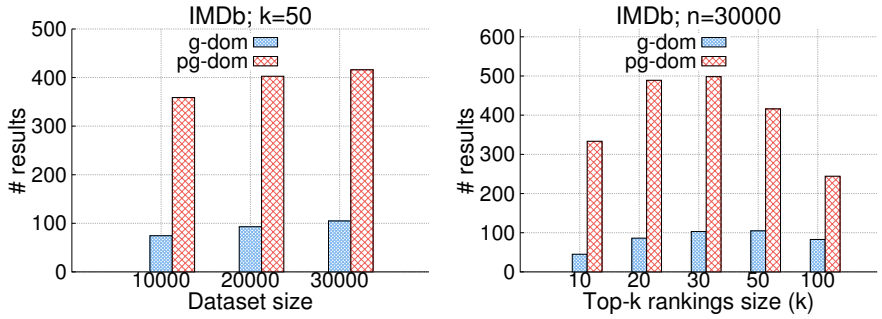
Method	Input size $\|I\|$	#results	#constraints	#rank-criteria
g-dom	1	466.4	43.5	13.0
	2	48.5	40.2	12.7
	3	7.4	20.1	10.8
	4	1.3	7.4	4.6
	5	0.2	1.6	1.3
pg-dom	1	466.4	43.5	13.0
	2	462.2	53.5	13.0
	3	424.6	59.9	13.0
	4	379.9	62.2	13.0
	5	346.7	65.5	13.0

Table 6.1: Results discovered with different input size

algorithm for partial group dominance pg-dom$\star$ continuously outperforms the other two methods with both datasets. For brevity, we will not go into the details of the results with group partial dominance. The results with the SYNT dataset show that g-dom$\star$ and pg-dom$\star$ manage to stay far below the interactive system response threshold of 500 ms, even with large number of rankings, while this is not the case with the other two methods.

We study the runtime for computing fractional group dominance with different threshold size θ, as presented in Figure 6.12. We observe that smaller values of θ reduce the performance of all methods, for both datasets. For instance, with the IMDb dataset, fg-dom$\star$ finishes computation on average in 163.2 ms for $\theta = 0.7$ and 90.1 ms for $\theta = 0.9$. With $\theta = 0.7$, fg-dom and fg-dom+ need 400.2 and 305.7 ms, respectively, while with $\theta = 0.9$ they finish processing in 376.6 and 191.5 ms, respectively. A smaller value of θ means being less restrictive when computing results, thus leading to more processing and later pruning of not dominating entities, which results in slower performance. For example, Al Pacino co-occurs in 90 rankings together with Hugh Jackman and with $\theta = 0.7$, in order for Pacino $\prec_\theta$ Jackman, he needs to be ranked higher in 63 of the 90 lists. This means that there can be 27 rankings where he is ranked lower than Jackson, i.e., it will be pruned from the possible results after finding it in lower position 27 times. On the other hand, with $\theta = 0.9$, Pacino can be pruned after being ranked lower in only 9 rankings, which results in faster execution time. Note that with non-fractional dominance, Pacino can be pruned if found ranked lower in only one ranking. Moreover, on the IMDb dataset, we observe that our top-m approach fg-dom$\star$ is faster than fg-dom by a factor of 4.2 and a factor of 2.1 to fg-dom+.

Results Discovered. Our system discovers dominating entities characterized by various constraints and ranking criteria, thus providing valuable insight about the input entities. We study the effect of the size of the input, dataset, and the size of the top-k lists over the number of all possible, correct dominating entities discovered over the IMDb dataset. Table 6.1 shows the average number of dominating entities, as well as the average number of distinct atomic constraints



(a) (b)

Figure 6.13: Results discovered with different
(a) dataset size and (b) top-k size

and ranking criteria found in the ranking lists which dominate the input entities. With group dominance, as expected, we observe that the number of results is significantly decreasing with larger input size, since each of the results need to co-occur with each of the input entities.

Figure 6.13 shows the average total number of dominating entities found, with different dataset and top-k size for the IMDb dataset. When the dataset grows, two conflicting situations occur that influence the number of results. On the one hand, more potential entities are introduced that could make it into the result. On the other hand, more rankings introduce a higher "risk" that an entity is not qualifying as a result since there might be a ranking where it is dominated by one of the input entities. In the figure, we see that g-dom returns on average 74.4 dominating entities when run on 10k rankings, while it computes 104.8 results when executed on 30k rankings, which is an increase of 41%. For pg-dom, the increase is less strong, but the absolute numbers were already quite high for 10k rankings. We further see in Figure 6.13(b) that for group dominance, the number of dominating entities increases with larger k and peaks for $k = 30$ before starting to decrease with $k = 50$ and $k = 100$. The decrease of number of results is due to the higher chance of eliminating a dominating entity with larger top-k rankings, considering that with longer lists the probability of occurring lower than the input is higher. We observe similar behavior with partial group dominance.

6.6.4 Lessons Learned

In summary, the main findings from the above experimental results are:

- The smart approach relying on probabilistic size estimation is able to pick the most promising approach to satisfy a user request and outperforms both the safe and naive approaches, for all choices of top-m results requested.

- The methods utilizing the prune-and-blacklist optimization are more efficient than their baseline equivalents, across all parameters examined: different input, dataset, and top-k rankings size.
- The top-m methods significantly outperform both the prune-and-blacklist and the baseline methods, with performance benefits up to two orders of magnitude over its counterparts.
- The top-m methods are more robust when processing more data, their runtime only mildly increases with larger input, dataset, or top-k rankings size.
- In computing fractional dominance, smaller values of the threshold θ reduce the performance of all methods. A smaller threshold means being less restrictive when computing results, thus leading to more processing and later pruning of not-dominating entities, which results in slower performance.
- Our approach always discovers dominating entities that are characterized by various constraints and ranking criteria, which provide valuable insight about the input entities.

6.7 Summary

This chapter presented COMPETE, an approach that enables access to and understanding of outstanding entities by computing a ranking-induced dominance relationship between them. We first defined the notion of dominance in different flavors using the position of competing entities in rankings, and generalized this for groups of input entities. COMPETE uses a probabilistic model in estimating result sizes, utilized for choosing the most promising and efficient approach in satisfying a user request. The core computational problem was optimized through index structures and early result eviction, demonstrating clear gains in the experimental evaluation over two datasets. A large volume of discovered results and their characteristic in the rankings provide valuable insight into the relative performance of the entities of interest in the dataset.

Chapter 7

Conclusion and Outlook

In this thesis we considered the problem of exploring data through ranked entities by reverse engineering top-k database queries and discovering dominant entities in rankings.

We proposed PALEO, an approach to reverse engineer top-k OLAP queries. PALEO mainly operates on subset of the base data, held in memory, and further uses data samples, histograms, and descriptive statistics to identify potentially valid queries. Further, we augmented the approach to discover queries that produce results similar—within a distance measure threshold—to the input list and presented techniques that incorporate distance-induced bounds used for early pruning of candidates. The main difficulty in identifying valid queries is to limit the number of false positives, that cause needless query validations, together with limiting the false negatives, which cause loss in recall. To this purpose, we proposed a probabilistic model that evaluates the suitability of candidate queries, which, combined with an iterative refinement of their validation, was proven to drastically decrease the amount of time to validate (or invalidate) queries. Analytics data is usually organized in complex schemas, containing many relations. Thus, we proposed a complementary approach to discover the join constraints of top-k join queries, utilizing an advanced classification system that prioritizes evaluation of promising candidate joins.

Furthermore, we have presented COMPETE, an approach that enables understanding of outstanding entities through computing a ranking-induced dominance relationship between them. We defined several notions of dominance which differ in computational complexity and strictness of the dominance concept, providing an intuitive perspective of the strengths and weaknesses of entities and their relative standing in the rankings. COMPETE employs a probabilistic model in estimating result sizes, used in picking the most promising approach to satisfy a user request at minimal runtime latency. We proposed a stack of algorithms to compute the various types of dominance, relying on suitable index structures and pruning techniques that avoid significant data access over large datasets of rankings.

PALEO currently reverse engineers top-k queries with a conjunction of equality predicates as selection condition in the queries. Extending the framework to deal with more expressive selection predicates that include disjunctions and inequality conditions could be considered for future work. Further, PALEO could handle more complex ranking criteria, allowing more arithmetical operations within the aggregation function, as well as support for statistical functions. Recently, crowdsourcing platforms that harness the crowd to infer top-k rankings have emerged. Frequently, the top-k rankings contain metadata that describe their properties. It would be interesting to reverse engineer these rankings and to discover the characteristics of their content through the description provided from real users.

Due to the sheer volume of possible outcomes with our COMPETE framework, a way to order results based on user-perceived, personalized relevance could be considered. Moreover, we could develop a model for probabilistic estimation of the result size with fractional dominance, based on the threshold θ. The model needs to consider how to choose the fraction of rankings θ in order to guarantee results to the user. In this way, the framework can efficiently satisfy user requests in scenarios when non-fractional dominance computes empty results.

Appendix A

Appendix

A.1 Sample Queries

Queries on a single relation:

q_avg01_1:

```
SELECT c_name, AVG(s_acctbal)
FROM tpch
WHERE r_name = 'MIDDLE EAST'
GROUP BY c_name
ORDER BY AVG(s_acctbal) DESC
LIMIT 5
```

q_avg01_2:

```
SELECT c_name, AVG(s_acctbal)
FROM tpch
WHERE r_name = 'MIDDLE EAST'
AND o_orderpriority = '1-URGENT'
GROUP BY c_name
ORDER BY AVG(s_acctbal) DESC
LIMIT 5
```

q_sum01_3:

```
SELECT c_name, AVG(s_acctbal)
FROM tpch
WHERE r_name = 'MIDDLE EAST'
AND o_orderpriority = '1-URGENT'
AND n_name = 'SAUDI ARABIA'
GROUP BY c_name
ORDER BY AVG(s_acctbal) DESC
LIMIT 5
```

q_max01_1:

```
SELECT c_name, MAX(l_extendedprice)
FROM tpch
WHERE c_mktsegment = 'HOUSEHOLD'
GROUP BY c_name
ORDER BY MAX(l_extendedprice) DESC
LIMIT 5
```

q_max01_2:

```
SELECT c_name, MAX(l_extendedprice)
FROM tpch
WHERE c_mktsegment = 'HOUSEHOLD'
AND l_shipinstruct = 'DELIVER IN PERSON'
GROUP BY c_name
ORDER BY MAX(l_extendedprice) DESC
LIMIT 5
```

q_max01_3:

```
SELECT c_name, MAX(l_extendedprice)
FROM tpch
WHERE c_mktsegment = 'HOUSEHOLD'
AND l_shipinstruct = 'DELIVER IN PERSON'
AND l_returnflag = 'R
GROUP BY c_name
ORDER BY MAX(l_extendedprice) DESC
LIMIT 5
```

q_sum01_1:

```
SELECT c_name, SUM(ps_supplycost)
FROM tpch
WHERE n_name = 'JAPAN'
GROUP BY c_name
ORDER BY SUM(ps_supplycost) DESC
LIMIT 5
```

q_sum01_2:

```
SELECT c_name, SUM(ps_supplycost)
FROM tpch
WHERE n_name = 'JAPAN'
AND p_container = 'JUMBO BAG'
GROUP BY c_name
ORDER BY SUM(ps_supplycost) DESC
LIMIT 5
```

q_sum01_3:

```
SELECT c_name, SUM(ps_supplycost)
FROM tpch
WHERE n_name = 'JAPAN'
AND p_container = 'JUMBO BAG'
AND l_shipmode = 'TRUCK'
GROUP BY c_name
ORDER BY SUM(ps_supplycost) DESC
LIMIT 5
```

q_sum2s01_1:

```
SELECT c_name,
SUM(o_totalprice+s_acctbal)
FROM tpch
WHERE l_shipmode = 'FOB'
GROUP BY c_name
ORDER BY
SUM(o_totalprice+s_acctbal) DESC
LIMIT 5
```

q_sum2s01_2:

```
SELECT c_name,
SUM(o_totalprice+s_acctbal)
FROM tpch
WHERE l_shipmode = 'FOB'
AND o_orderpriority = '1-URGENT'
GROUP BY c_name
ORDER BY
SUM(o_totalprice+s_acctbal) DESC
LIMIT 5
```

q_sum2s01_3:

```
SELECT c_name,
SUM(o_totalprice+s_acctbal)
FROM tpch
WHERE l_shipmode = 'FOB'
AND o_orderpriority = '1-URGENT'
AND l_shipinstruct = 'TAKE BACK RETURN'
GROUP BY c_name
ORDER BY
SUM(o_totalprice+s_acctbal) DESC
LIMIT 5
```

q_noagg01_1:

```
SELECT c_name, o_totalprice
FROM tpch
WHERE p_brand = 'Brand#54'
GROUP BY c_name
ORDER BY o_totalprice DESC
LIMIT 5
```

q_noagg01_2:

```
SELECT c_name, o_totalprice
FROM tpch
WHERE p_brand = 'Brand#54'
AND p_container = 'LG BAG'
GROUP BY c_name
ORDER BY o_totalprice DESC
LIMIT 5
```

q_noagg01_3:

```
SELECT c_name, o_totalprice
FROM tpch
WHERE p_brand = 'Brand#54'
AND p_container = 'LG BAG'
AND n_name = 'ARGENTINA'
GROUP BY c_name
ORDER BY o_totalprice DESC
LIMIT 5
```

q_sum2p01_1:

```
SELECT c_name,
SUM(ps_supplycost*ps_availqty)
FROM tpch
WHERE n_name = 'JAPAN'
GROUP BY c_name
ORDER BY
SUM(ps_supplycost*ps_availqty) DESC
LIMIT 5
```

q_sum2p01_2:

```
SELECT c_name,
SUM(ps_supplycost*ps_availqty)
FROM tpch
WHERE n_name = 'JAPAN'
AND p_container = 'JUMBO BAG'
GROUP BY c_name
ORDER BY
SUM(ps_supplycost*ps_availqty) DESC
LIMIT 5
```

q_sum2p01_3:

```
SELECT c_name,
SUM(ps_supplycost*ps_availqty)
FROM tpch
WHERE n_name = 'JAPAN'
AND p_container = 'JUMBO BAG'
AND l_shipmode = 'TRUCK'
GROUP BY c_name
ORDER BY
SUM(ps_supplycost*ps_availqty) DESC
LIMIT 5
```

Queries with joins:

jq_avg01_1:

```
SELECT o_clerk AS entity,
AVG(l_extendedprice)
FROM orders,
lineitem
WHERE o_orderkey = l_orderkey
AND o_orderstatus = F'
GROUP BY entity
ORDER BY AVG(l_extendedprice) DESC
LIMIT 5
```

jq_sum01_0:

```
SELECT s_name AS entity,
SUM(ps_supplycost)
FROM partsupp,
supplier
WHERE s_suppkey = ps_suppkey
GROUP BY entity
ORDER BY SUM(ps_supplycost) DESC
LIMIT 5
```

jq_sum2s01_0:

```
SELECT n_name AS entity,
SUM(ps_supplycost+ps_availqty)
FROM supplier,
partsupp,
nation
WHERE s_suppkey = ps_suppkey
AND n_nationkey = s_nationkey
GROUP BY entity
ORDER BY
SUM(ps_supplycost+ps_availqty) DESC
LIMIT 5
```

jq_avg03_3:

```
SELECT c_name AS entity,
AVG(o_totalprice)
FROM orders,
customer,
nation
region
WHERE c_custkey = o_custkey
AND n_nationkey = c_nationkey
AND n_regionkey = r_regionkey
AND c_mktsegment = 'FURNITURE'
AND n_name = 'GERMANY'
AND o_orderstatus = 'F'
GROUP BY entity
ORDER BY AVG(o_totalprice) DESC
LIMIT 5
```

jq_max01_2:

```
SELECT c_name AS entity,
MAX(o_totalprice)
FROM orders,
customer
WHERE c_custkey = o_custkey
AND c_mktsegment = 'FURNITURE'
AND o_orderstatus = 'F'
GROUP BY entity
ORDER BY MAX(o_totalprice) DESC
LIMIT 5
```

jq_noagg01_2:

```
SELECT o_clerk AS entity,
l_extendedprice
FROM orders,
lineitem,
customer
WHERE o_orderkey = l_orderkey
AND c_custkey = o_custkey
AND o_orderstatus = 'F'
AND c_mktsegment = 'FURNITURE'
ORDER BY l_extendedprice DESC
LIMIT 5
```

jq_sum2p01_2:

```
SELECT o_clerk AS entity,
SUM(l_quantity*l_extendedprice)
FROM orders,
lineitem,
customer
WHERE o_orderkey = l_orderkey
AND c_custkey = o_custkey
AND o_orderstatus = 'F'
AND c_mktsegment = 'FURNITURE'
GROUP BY entity
ORDER BY
SUM(l_quantity*l_extendedprice) DESC
LIMIT 5
```

jq_max03_2:

```
SELECT s_name AS entity,
MAX(ps_supplycost)
FROM supplier,
partsupp,
part,
nation
WHERE s_suppkey = ps_suppkey
AND ps_partkey = p_partkey
AND s_nationkey = n_nationkey
AND p_container = 'JUMBO PACK'
AND n_name = 'GERMANY'
GROUP BY entity
ORDER BY MAX(ps_supplycost) DESC
LIMIT 5
```

A.2 Sample Sets of Input Entities

Kurt Russell; Mel Gibson; Tom Cruise; Jason Isaacs; Gael García Bernal
Gael García Bernal; Jason Isaacs; Tom Cruise; Mel Gibson; Mickey Rourke
Mel Gibson; Brad Pitt; Mickey Rourke; Joe Pantoliano; Rachel McAdams
Rachel McAdams; Joe Pantoliano; Mickey Rourke; Brad Pitt; Tom Cruise
Kevin Spacey; Daniel Day-Lewis; Brad Pitt; Al Pacino; Edward Norton
Edward Norton; Al Pacino; Brad Pitt; Daniel Day-Lewis; George Clooney
Johnny Depp; George Clooney; Jake Gyllenhaal; Mark Wahlberg; Sean Penn
Sean Penn; Mark Wahlberg; Jake Gyllenhaal; George Clooney; Mel Gibson
Kim Basinger; Kathleen Turner; Imogene Coca; Annie McEnroe; Angelina Jolie
Angelina Jolie; Annie McEnroe; Imogene Coca; Kathleen Turner; Julianne Moore
Nicole Kidman; Angelina Jolie; Cate Blanchett; Julianne Moore; Meryl Streep
Meryl Streep; Julianne Moore; Cate Blanchett; Angelina Jolie; Nicole Kidman
Jude Law; Ewan McGregor; Liam Neeson; Arnold Schwarzenegger; Gene Hackman
Gene Hackman; Arnold Schwarzenegger; Liam Neeson; Ewan McGregor; Kevin Bacon
John Travolta; Alec Baldwin; Kevin Costner; Denzel Washington; Kevin Bacon
Kevin Bacon; Denzel Washington; Kevin Costner; Alec Baldwin; Ed Harris
Samuel L. Jackson; Harrison Ford; Jeff Bridges; Russell Crowe; Ed Harris
Ed Harris; Russell Crowe; Jeff Bridges; Harrison Ford; Bill Paxton
Cameron Diaz; Angelina Jolie; Cate Blanchett; Julia Roberts; Sigourney Weaver
Sigourney Weaver; Julia Roberts; Cate Blanchett; Angelina Jolie; Drew Barrymore
Tim Robbins; Pierce Brosnan; Sean Penn; John Cusack; Bill Paxton
Bill Paxton; John Cusack; Sean Penn; Pierce Brosnan; Dustin Hoffman
Michael Douglas; Richard Gere; Dennis Quaid; Bill Murray; Dustin Hoffman
Dustin Hoffman; Bill Murray; Dennis Quaid; Richard Gere; Alec Baldwin
Harrison Ford; Michael Douglas; Samuel L. Jackson; Arnold Schwarzenegger; Alec Baldwin
Alec Baldwin; Arnold Schwarzenegger; Samuel L. Jackson; Michael Douglas; Richard Gere
Jason Statham; Hugh Jackman; Jean-Claude Van Damme; Jake Gyllenhaal; Joaquin Phoenix
Joaquin Phoenix; Jake Gyllenhaal; Jean-Claude Van Damme; Hugh Jackman; Owen Wilson
Matthew McConaughey; Forest Whitaker; Colin Firth; Woody Harrelson; Tom Wilkinson
Tom Wilkinson; Woody Harrelson; Colin Firth; Forest Whitaker; Vin Diesel

List of Figures

List of Algorithms

List of Tables

Bibliography

[ACD02] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In Rakesh Agrawal and Klaus R. Dittrich, editors, *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 5–16. IEEE Computer Society, 2002.

[ARSZ03] Giuseppe Amato, Fausto Rabitti, Pasquale Savino, and Pavel Zezula. Region proximity in metric spaces and its use for approximate similarity search. *ACM Trans. Inf. Syst.*, 21(2):192–227, 2003.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994.

[AtCKT11] Bogdan Alexe, Balder ten Cate, Phokion G. Kolaitis, and Wang Chiew Tan. Designing and refining schema mappings via data examples. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 133–144. ACM, 2011.

[BCC10] Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval - Implementing and Evaluating Search Engines*. MIT Press, 2010.

[BCS14] Angela Bonifati, Radu Ciucanu, and Slawek Staworko. Interactive inference of join queries. In Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy, editors, *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014.*, pages 451–462. OpenProceedings.org, 2014.

[BCS16] Angela Bonifati, Radu Ciucanu, and Slawek Staworko. Learning join queries from user examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38, 2016.

[BCT06] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for DBMS testing. *IEEE Trans. Knowl. Data Eng.*, 18(12):1721–1725, 2006.

[BFO+09] Christian Böhm, Frank Fiedler, Annahita Oswald, Claudia Plant, and Bianca Wackersreuther. Probabilistic skyline queries. In David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy J. Lin, editors, *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 651–660. ACM, 2009.

[BJK+12] Lukas Blunschi, Claudio Jossen, Donald Kossmann, Magdalini Mori, and Kurt Stockinger. SODA: generating SQL for business users. *PVLDB*, 5(10):932–943, 2012.

[BKL07] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 506–515. IEEE Computer Society, 2007.

[BKLÖ07] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. Qagen: generating query-aware test databases. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 341–352. ACM, 2007.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In Dimitrios Georgakopoulos and Alexander Buchmann, editors, *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 421–430. IEEE Computer Society, 2001.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986.*, pages 61–71. ACM Press, 1986.

[CBC+00] Yuan-Chi Chang, Lawrence D. Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: Indexing for linear optimization queries. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management*

of Data, May 16-18, 2000, Dallas, Texas, USA., pages 391–402. ACM, 2000.

[CGGL03] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 717–719. IEEE Computer Society, 2003.

[Chv79] Vasek Chvátal. A greedy heuristic for the set-covering problem. *Math. Oper. Res.*, 4(3):233–235, 1979.

[CJT+06] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. Finding k-dominant skylines in high dimensional space. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 503–514. ACM, 2006.

[CL08] Lei Chen and Xiang Lian. Dynamic skyline queries in metric spaces. In Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu, editors, *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, volume 261 of *ACM International Conference Proceeding Series*, pages 333–343. ACM, 2008.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[CM17] Paolo Ciaccia and Davide Martinenghi. Reconciling skyline and ranking queries. *PVLDB*, 10(11):1454–1465, 2017.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[Dat] Database Basketball portal. `http://www.databasebasketball.com`, Retrieved April 12, 2016.

[DBL] DBLP computer science bibliography. `https://dblp.uni-trier.de`, Retrieved April 12, 2016.

[DP13] Marina Drosou and Evaggelia Pitoura. Ymaldb: exploring relational databases via result-driven recommendations. *VLDB J.*, 22(6):849–874, 2013.

[DPD14] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: an automatic query steering framework for interactive data exploration. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 517–528. ACM, 2014.

[DPD16] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. AIDE: an active learning-based approach for interactive data exploration. *IEEE Trans. Knowl. Data Eng.*, 28(11):2842–2856, 2016.

[FKM+04] Ronald Fagin, Ravi Kumar, Mohammad Mahdian, D. Sivakumar, and Erik Vee. Comparing and aggregating rankings with ties. In Catriel Beeri and Alin Deutsch, editors, *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 14-16, 2004, Paris, France*, pages 47–58. ACM, 2004.

[FKS03] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.

[FLN01] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In Peter Buneman, editor, *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA*. ACM, 2001.

[GM15] Benoît Groz and Tova Milo. Skyline queries with noisy comparisons. In Tova Milo and Diego Calvanese, editors, *Proceedings of the 34th ACM Symposium on Principles of Database Systems, PODS 2015, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 185–198. ACM, 2015.

[GR12] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east, IDC - EMC Corporation, 2012. `www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf`, Retrieved November 6, 2018.

[GS10] Georg Gottlob and Pierre Senellart. Schema mapping discovery from data instances. *J. ACM*, 57(2):6:1–6:37, 2010.

[HKP01] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In Sharad Mehrotra and Timos K. Sellis, editors,

Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001, pages 259–270. ACM, 2001.

[HM03] Sven Helmer and Guido Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *VLDB J.*, 12(3):244–261, 2003.

[IAE04] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004.

[IBS08] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.

[IMS13] Evica Ilieva, Sebastian Michel, and Aleksandar Stupar. The essence of knowledge (bases) through entity rankings. In Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi, editors, *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, pages 1537–1540. ACM, 2013.

[Int] Internet Movie Database. `https://www.imdb.com/interfaces/`, Retrieved April 12, 2016.

[IPC15] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 277–281. ACM, 2015.

[Jac12] Paul Jaccard. The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50, Feb 1912.

[JGP15] Manas Joglekar, Hector Garcia-Molina, and Aditya G. Parameswaran. Smart drill-down: A new data exploration operator. *PVLDB*, 8(12):1928–1931, 2015.

[JGP16] Manas Joglekar, Hector Garcia-Molina, and Aditya 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. IEEE Computer Society, 2016.

[JN15] Lilong Jiang and Arnab Nandi. Snaptoquery: Providing interactive feedback during exploratory query specification. *PVLDB*, 8(11):1250–1261, 2015.

[KÇZ14] Alexander Kalinin, Ugur Çetintemel, and Stanley B. Zdonik. Interactive data exploration using semantic windows. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 505–516. ACM, 2014.

[KÇZ15] Alexander Kalinin, Ugur Çetintemel, and Stanley B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.

[Ken38] Maurice G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[KHP10] Abhijith Kashyap, Vagelis Hristidis, and Michalis Petropoulos. Facetor: cost-driven exploration of faceted query results. In Jimmy Huang, Nick Koudas, Gareth J. F. Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An, editors, *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, pages 719–728. ACM, 2010.

[Kim96] Ralph Kimball. *The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses*. John Wiley, 1996.

[KJTN14] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. Distributed and interactive cube exploration. In Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 472–483. IEEE Computer Society, 2014.

[KLP75] H. T. Kung, Fabrizio Luccio, and Franco P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[KLS18] Dmitri V. Kalashnikov, Laks V. S. Lakshmanan, and Divesh Srivastava. Fastqre: Fast query reverse engineering. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 337–350. ACM, 2018.

[KML08] Mohamed E. Khalefa, Mohamed F. Mokbel, and Justin J. Levandoski. Skyline query processing for incomplete data. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 556–565. IEEE Computer Society, 2008.

[KRR02] Donald Kossmann, Frank Ramsak, and Steffen Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases, August 20-23, 2002, Hong Kong, China*, pages 275–286. Morgan Kaufmann, 2002.

[KSA14] Hina A. Khan, Mohamed A. Sharaf, and Abdullah Albarrak. Divide: efficient diversification for interactive data exploration. In Christian S. Jensen, Hua Lu, Torben Bach Pedersen, Christian Thomsen, and Kristian Torp, editors, *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, pages 15:1–15:12. ACM, 2014.

[LCI06] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 61–72. ACM, 2006.

[LCIS05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. Ranksql: Query algebra and optimization for relational top-k queries. In Fatma Özcan, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 131–142. ACM, 2005.

[LCM15] Hao Li, Chee-Yong Chan, and David Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13):2158–2169, 2015.

[LRB09] Marie-Jeanne Lesot, Maria Rifqi, and Hamid Benhadda. Similarity measures for binary and numerical data: a survey. *IJKESDP*, 1(1):63–84, 2009.

[MAM15] Evica Milchevski, Avishek Anand, and Sebastian Michel. The sweet spot between inverted indices and metric-space indexing for top-k-list similarity search. In Gustavo Alonso, Floris Geerts, Lucian Popa, Pablo Barceló, Jens Teubner, Martín Ugarte, Jan Van den Bussche, and Jan Paredaens, editors, *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 253–264. OpenProceedings.org, 2015.

[MC11] Denis Mindolin and Jan Chomicki. Preference elicitation in prioritized skyline queries. *VLDB J.*, 20(2):157–182, 2011.

[MHH00] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel,

Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 77–88. Morgan Kaufmann, 2000.

[MKZ08] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 499–510. ACM, 2008.

[MLVP17] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. New trends on exploratory methods for data analytics. *PVLDB*, 10(12):1977–1980, 2017.

[MQM97] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA.*, pages 100–111. ACM Press, 1997.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[PDCC15] Fotis Psallidas, Bolin Ding, Kaushik Chakrabarti, and Surajit Chaudhuri. S4: top-k spreadsheet-style search for query discovery. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 2001–2016. ACM, 2015.

[PJLY07] Jian Pei, Bin Jiang, Xuemin Lin, and Yidong Yuan. Probabilistic skylines on uncertain data. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 15–26. ACM, 2007.

[PM16] Kiril Panev and Sebastian Michel. Reverse engineering top-k database queries with PALEO. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux,*

France, March 15-16, 2016., pages 113–124. OpenProceedings.org, 2016.

[PM18] Kiril Panev and Sebastian Michel. Exploring pros and cons of ranked entities with compete. In *Proceedings of the ExploreDB'18, Houston, TX, USA, June 15, 2018*, 2018.

[PMM16] Kiril Panev, Evica Milchevski, and Sebastian Michel. Computing similar entity rankings via reverse engineering of top-k database queries. In *32nd IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016*, pages 181–188. IEEE Computer Society, 2016.

[PMMP16] Kiril Panev, Sebastian Michel, Evica Milchevski, and Koninika Pal. Exploring databases via reverse engineering ranking queries with PALEO. *PVLDB*, 9(13):1525–1528, 2016.

[PO] Xuedong Chen Pat O'Neil, Betty O'Neil. Star schema benchmark. `http://www.odbms.org/2014/03/star-schema-benchmark`, Retrieved August 15, 2015.

[PTFS03] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. An optimal and progressive algorithm for skyline queries. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 467–478. ACM, 2003.

[PTFS05] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005.

[PWM17] Kiril Panev, Nico Weisenauer, and Sebastian Michel. Reverse engineering top-k join queries. In Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*, volume P-265 of *LNI*, pages 61–80. GI, 2017.

[RDS07] Christopher Ré, Nilesh N. Dalvi, and Dan Suciu. Efficient top-k query evaluation on probabilistic data. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 886–895. IEEE Computer Society, 2007.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[RTG98] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. A metric for distributions with applications to image databases. In *ICCV*, pages 59–66, 1998.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[SCC+14] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 493–504. ACM, 2014.

[SIC07] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. Top-k query processing in uncertain databases. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 896–905. IEEE Computer Society, 2007.

[SK13] Thibault Sellam and Martin L. Kersten. Meet charles, big data query advisor. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org, 2013.

[SKB11] Lefteris Sidirourgos, Martin L. Kersten, and Peter A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 296–301. www.cidrdb.org, 2011.

[Spe94] Charles Spearmn. The american journal of psychology. *ACM Trans. Database Syst.*, 15(1):72–101, 1094.

[SPGW10] Anish Das Sarma, Aditya G. Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In Luc Segoufin, editor, *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, ACM International Conference Proceeding Series, pages 89–103. ACM, 2010.

[SS94] Arun N. Swami and K. Bernhard Schiefer. On the estimation of join result sizes. In Matthias Jarke, Janis A. Bubenko Jr., and Keith G. Jeffery, editors, *Advances in Database Technology - EDBT'94. 4th International Conference on Extending Database Technology, Cambridge, United Kingdom, March 28-31, 1994, Proceedings*, volume 779 of *Lecture Notes in Computer Science*, pages 287–300. Springer, 1994.

[TCP09] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *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. ACM, 2009.

[TCP14] Quoc Trung Tran, Chee Yong Chan, and Srinivasan Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5):721–746, 2014.

[TEO01] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 301–310. Morgan Kaufmann, 2001.

[TL08] Sandeep Tata and Guy M. Lohman. SQAK: doing more with keywords. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 889–902. ACM, 2008.

[TPC] TPC. TPC benchmarks. `http://www.tpc.org`, Retrieved August 15, 2015.

[TPM11] Eleftherios Tiakas, Apostolos N. Papadopoulos, and Yannis Manolopoulos. Progressive processing of subspace dominating queries. *VLDB J.*, 20(6):921–948, 2011.

[TSK05] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.

[TSW05] Martin Theobald, Ralf Schenkel, and Gerhard Weikum. An efficient and versatile query engine for topx search. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 625–636. ACM, 2005.

[TZES17] Wei Chit Tan, Meihui Zhang, Hazem Elmeleegy, and Divesh Srivastava. Reverse engineering aggregation queries. *PVLDB*, 10(11):1394–1405, 2017.

[Vaz01] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.

[VDKN10] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. Reverse top-k queries. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard

Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 365–376. IEEE Computer Society, 2010.

[WC17] Yaacov Y. Weiss and Sara Cohen. Reverse engineering spj-queries from examples. In Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 151–166. ACM, 2017.

[WCB17] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 452–466. ACM, 2017.

[YM07] Man Lung Yiu and Nikos Mamoulis. Efficient processing of top-k dominating queries on multi-dimensional data. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 483–494. ACM, 2007.

[ZEPS13] Meihui Zhang, Hazem Elmeleegy, Cecilia M. Procopiuc, and Divesh Srivastava. Reverse engineering complex join queries. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 809–820. ACM, 2013.

[ZGH+17] Kaiqi Zhang, Hong Gao, Xixian Han, Zhipeng Cai, and Jianzhong Li. Probabilistic skyline on incomplete data. In Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li, editors, *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*, pages 427–436. ACM, 2017.

[ZHC+06] Zhen Zhang, Seung-won Hwang, Kevin Chen-Chuan Chang, Min Wang, Christian A. Lang, and Yuan-Chi Chang. Boolean + ranking: querying a database by k-constrained optimization. In Surajit

Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 359–370. ACM, 2006.

[ZLOT10] Zhenjie Zhang, Hua Lu, Beng Chin Ooi, and Anthony K. H. Tung. Understanding the meaning of a shifted sky: a general framework on extending skyline query. *VLDB J.*, 19(2):181–201, 2010.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

Kiril Panev

Education

03/2014 – 04/2019 **PhD in Computer Science**, *TU Kaiserslautern (11/2014 –), Germany, International Max Planck Research School for Computer Science (IMPRS-CS Scholarship), Saarland University (03/2014 – 10/2014), Germany.*
Thesis: Exploring Data through Ranked Entities.

10/2011 – 12/2013 **Masters in Computer Science**, *Saarland University, Saarbrücken, Germany*, GPA: 1.2 out of 1.0.
Thesis: Phrase Querying with Multiple Indexes.
Grade of Master's Thesis: 1.0 out of 1.0

10/2003 – 05/2008 **Bachelor of Computer Science**, *Faculty of Natural Sciences and Mathematics - Institute of Informatics, University "St. Cyril and Methodius", Skopje, Macedonia*, GPA: 8.55/10.
Thesis: Network Management - Main Prerequisite for Implementing Service Level Agreements.
Grade of Bachelor's Thesis: 10 out of 10

Work Experience

10/2011 – 02/2014 **Part Time Research Assistant (HiWi)**, *MPII Saarbrücken, Germany.*

04/2009 – 09/2011 **Data Warehouse Administrator**, *T-Mobile Macedonia, Skopje, Macedonia.*
Main activities and responsibilities:

- ETL procedures utilizing PL/SQL
- Data marts design and deployment
- Database design for new systems
- Administration of ETL processes

12/2007 – 04/2009 **Professional Services**, *EuroComputer Systems, Skopje, Macedonia.*
Main activities and responsibilities:

- System integration
- Database design
- Provision of technical support

Languages

Macedonian	Native
English	Level C2 by CEFR
German	Level B2 by CEFR