Algorithms for Continuous Queries: A Geometric Approach

by

Albert Yu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Pankaj K. Agarwal, Co-supervisor

_____
Jun Yang, Co-supervisor

_____
Ashwin Machanavajjhala

_____
Romit Roy Choudhury

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2013

ABSTRACT

Algorithms for Continuous Queries: A Geometric Approach

by

Albert Yu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Pankaj K. Agarwal, Co-supervisor

_____
Jun Yang, Co-supervisor

_____
Ashwin Machanavajjhala

_____
Romit Roy Choudhury

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2013

Abstract

There has been an unprecedented growth in both the amount of data and the number of users interested in different types of data. Users often want to keep track of the data that match their interests over a period of time. A continuous query, once issued by a user, maintains the matching results for the user as new data (as well as updates to the existing data) continue to arrive in a stream. However, supporting potentially millions of continuous queries is a huge challenge. This dissertation addresses the problem of scalably processing a large number of continuous queries over a wide-area network.

Conceptually, the task of supporting distributed continuous queries can be divided into two components—event processing (computing the set of affected users for each data update) and notification dissemination (notifying the set of affected users). The first part of this dissertation focuses on event processing. Since interacting with large-scale data can easily frustrate and overwhelm the users, top-k queries have attracted considerable interest from the database community as they allow users to focus on the top-ranked results only. However, it is nearly impossible to find a set of common top-ranked data that everyone is interested in, therefore, users are allowed to specify their interest in different forms of preferences, such as personalized ranking function and range selection. This dissertation presents geometric frameworks, data structures, and algorithms for answering several types of preference queries efficiently. Experimental evaluations show that our approaches outperform the previous ones by orders of magnitude.

The second part of the dissertation presents comprehensive solutions to the problem

of processing and notifying a large number of continuous range top-k queries across a wide-area network. Simple solutions include using a content-driven network to notify all continuous queries whose ranges contain the update (ignoring top-k), or using a server to compute only the affected continuous queries and notifying them individually. The former solution generates too much network traffic, while the latter overwhelms the server. This dissertation presents a geometric framework which allows the set of affected continuous queries to be described succinctly with messages that can be efficiently disseminated using content-driven networks. Fast algorithms are also developed to reformulate each update into a set of messages whose number is provably optimal, with or without knowing all continuous queries.

The final component of this dissertation is the design of a wide-area dissemination network for continuous range queries. In particular, this dissertation addresses the problem of assigning users to servers in a wide-area content-based publish/subscribe system. A good assignment should consider both users' interests and locations, and balance multiple performance criteria including bandwidth, delay, and load balance. This dissertation presents a Monte Carlo approximation algorithm as well as a simple greedy algorithm. The Monte Carlo algorithm jointly considers multiple performance criteria to find a broker-subscriber assignment and provides theoretical performance guarantees. Using this algorithm as a yardstick, the greedy algorithm is also concluded to work well across a wide range of workloads.

Contents

List of Tables

List of Figures

1

Introduction

This is the age of data. We are witnessing the unprecedented growth in both the amount of data and the number of users interested in different types of data. When users look for data that are relevant to their interests, each of their data requests is generally expressed as a *query* to a database. To process a query, the database performs a sequence of operations on the data and returns relevant answers to the query. The growth of data, however, brings new challenges for efficient query processing.

The first challenge is to design algorithms that achieve fast response time. At any point in time, thousands of user queries can be posed against a snapshot of a large database through interactive interfaces. With the sheer volume of available data, millions of data may be relevant to each query. A naive approach may take minutes to answer each query, but typically users want to receive the answers immediately. For example, it was reported that delay is one of the primiary sources of web users' frustration [37] and that 4 seconds is all an average online shopper will wait for before potentially abondoning a retail site [14]. As another example, financial services need to provide real-time finanical information to their clients. Consider an investor who wants to identify profitable trades in a stock market. Since the market conditions can change in a matter of seconds, an investor may miss

buying/selling opportunities if stock tickers are not received within seconds. Obtaining answers to the queries in a timely manner is a critical challenge for efficient query processing.

The second challenge for query processing is that in addition to fast response time, the query answers need to be of high quality. It is not enough to simply return all relevant answers, because users would get overwhelmed by the sheer volume of query answers. This often frustrates the users and deteriorates the query answers to the point of uselessness. Fortunately, returning *all* relevant answers is unnecessary for most applications; users are only interested to know a limited set of top-ranked relevant answers. For example, Google News prioritize the stories and cluster similar new articles together. This design allows users to easily catch the news breaking stories and skim through the top news of the day. Undoubtedly, allowing users to focus on the top-ranked relevant data is key to ensuring high quality of results.

As the number of users continues to increase at an astounding rate, finding a set of common top-ranked data that everyone is interested in is nearly impossible. Since every user has different interests and preferences for ranking data that match her interests, high-quality query processing systems have to provide personalized results for user queries. For example, a stock screener may list stocks with a wide range of numeric attributes, e.g., market capitalization, trade volume, price-to-earning ratio, etc. Representing different user interests, user queries may have different range conditions. One user may be interested in a stock only if its market capitalization is at least 500 million, while another user may specify other range conditions on other attributes. In addition, the stocks that satisfy a user's range conditions are ranked according to her preference, which depends on whether the list of stocks identifies buying or selling opportunities, and will vary according to ones personal investing style and tolerance for risk. As another example, in online sports communities, sport fans share their opinions on players'performances with each other. Many of them like to analyze players with respect to customized performance metrics, e.g., for NBA

fans, a user query can be "Return top-10 players with the highest true shooting percentage (TS%) who have at least 2 steals and 5 rebounds," or "Return top-10 players with the most field goals whose field goal percentage is at least 45% and defensive rating is at most 102." Today, cameras hang in a handful of NBA arenas are able to track every player on court and record every move 25 times per second [109]. The improvement on optical tracking techniques creates not only lots of data events, but also lots of performance attributes, such as speed/distance, passes made/received, individual touches, etc [109]. A big challenge is to support queries on these high-dimensional data in a scalable way. Hypothetically speaking, if a web-based NBA search engine allows millions of the NBA fans around the world to query these high-dimensional data simultanenously, a more rigorous algorithmic approach is needed for query processing in order to keep up with the growth of users.

Continuous query. Very often, users want to keep track of the data that match their interests over a period of time. The answers to these continuous user queries continue to be updated as *events* (data insertion, deletion, and update) keep arriving in a stream. Traditionally, users poll sources for information. However, users may miss important events because those important events may arrive at any point in time. In addition, frequently polling for updates is hardly scalable for many applications. Alternatively, the publish/subscribe model, which pushes notifications to users with matching interests, expressed as subscriptions [1], is better suited for ensuring scalability and timely delivery of information. Even if events do not come at a very high rate, processing and pushing them to the servers for affected queries at their published time improves query response time, because answers to the queries can be found by simply retrieving the pre-computed results. Furthermore, the quality of the answers is also improved because a better, but more costly, algorithm can be used to precompute the answers. Supporting for a large-scale set of subscriptions is important in many application domains, including personal, commerical, and

[1] In this dissertation, we will use "continuous query" and "subscription" interchangeably.

security, etc. For example,

Portfolio monitoring. Financial services provide financial updates such as stock tickers to their clients in real-time.

Web alerts. Instead of repeatedly querying a search engine with the same set of search terms, web monitoring systems such as Google Alerts automatically notify users of the latest relevant new contents (from web, blogs, news, etc) based on users' search queries. Users can use the alerts to follow a developing news story, get the latest news on a celebrity or their favorite sports teams, etc.

Web page recommendations. Web search engines, such as Google, automatically detect standing interests of their users from their search logs [119]. When new contents match the users' interests, they are presented to the users as recommendations.

Content delivery services. Content delivery services such as Akamai employ extensive caching of database query results at their "edge servers" to improve performance. These caches need to be kept up-to-date when the central database is updated.

Social annotation of news. Social updates, e.g. tweets, on news events often reflect public views of those events. They are nicely complementary to news articles written by professional journalists. To automatically annotating news stories with social updates at a news website, news stories are treated as subscriptions and tweets are treated as data events [106].

Social networks. In social networking websites, such as Facebook, users see a constantly updated list of recent activity of their friends. Here, each user subscribes to events from her friends who act as event publishers.

E-Commerce. Online aution and shopping sites such as eBay provides subscription services for their customers to stay up-to-date with new and modified matching items.

Network security. Internet Service Providers monitor network traffic in real-time at various routers to detect possible network attacks.

As in the case of snapshot queries, data is ranked based on personal preferences, and subscriptions only receive top-ranked events that match their interests. For example, in the case of portfolio monitoring, stock price updates may match a large number of subscriptions. While many users may be interested in the same stock, each user may specify a unique set of contraints. One user may want to be notified of a stock update only if its price-to-earning ratio is at least 20 while another user may want to get a stock update only if its debt-to-assets ratio is at most 0.1. Since users want to be notified only when certain customized conditions are met, the data needs are potentially different across users.

For applications that require low frequency of notification, query results can be batched after they have been gathered over a period of time. For example, Google Alerts collects at least thousands of events that match a subscription every day. If all those events are sent to a user's email inbox at their occurring times, the user will most likely mark all those messages as spams and unsubscribe to Google Alerts. Therefore, Google Alerts provide options for users to choose the freqeuncy of receiving batch results and to receive only the best results in each batch.

1.1 Challenges

Supporting a large-scale set of subscriptions is challenging for many reasons:

Diversity of interests and personal preferences. Given an event update, how can we quickly find the (small) set of users that need to be notified, in the presence of potentially millions of subscriptions? A brute-force approach that scans through the whole set of subscriptions does not scale. Second, every user has different interests and personal preferences. The flexibility of user preferences together with the diversity of user interests demand more

powerful event processing funtionalities, making the task of event processing much more difficult. At the same time, for applications such as portfolio monitoring, the stock updates must reach relevant users in timely fashion. It is inefficient to support flexible user preferences in two phrases—1) computing the set of matching subscriptions, and 2) testing the notification conditions of each matching subscription individually. It is because the number of matching subscriptions can be significantly larger than the number of matching subscriptions whose notification conditions are met, i.e., many matching subscriptions need not to be notified. The challenge lies in finding a way to group processing subscriptions despite the diversity of interests and personal preferences.

Joint optimization of event processing and notification dissemination. An even more challenging application setting is when a large number of users are located across a wide-area network. In this setting, each user maintains a list of top-ranked objects locally. For each event updating the database, we must notify all subscriptions whose lists are affected. Notification messages should carry enough information so that the affected subscriptions can update their top-ranked lists accordingly.

A naive approach would be to use a central server to compute the list of users who needed to be notified and then unicast updates to each of them. However, the server could become a bottleneck with processing and messaging costs at least linear in the number of affected users. Alternatively, a publish/subscribe system can be leveraged as a means for distributing the event updates to the users. A publish/subscribe system typically employs a network of *brokers* that serve as the middleware between the data providers and users. Traditionally, subscriptions are stateless: they can be processed by only examining the incoming event itself. Personalized results, however, often require stateful subscriptions: whether a subscription is affected depends on how the updated objects ranks against others that also satisfy the selection constraints. A straightforward solution is to add post-processing logic and maintain additional information on the user side, but one

has to leverage both ranking and selection criteria in order to reduce network traffic.

Need for a well-designed dissemination network. Last but not least, a well-designed publish/subscribe network is key to ensuring efficient event processing and notification dissemination. A particular problem of interest is how to assign users to brokers, such that event processing and notification dissemination are jointly optimized. Intuitively, it is beneficial to assign subscriptions with similar interests to the same broker, because events delivered to the broker serve multiple subscriptions, potentially saving communication. On the other hand, we need to be careful in letting one broker handle users that are far away in terms of network distance, because doing so may violate delivery latency requirements and increase communication costs. Balancing the two considerations—similarity of interests in the event space and proximity of locations in the network space—is a hard optimization. The optimal trade-off between the two also depends on the amounts of events matching shared versus disjoint interests. Therefore, the best solution for a given system must take into account subscription interests and locations as well as event distributions.

1.2 Data Model and User Queries

Conceptually, all data of interest can be modeled as a relational database. In this dissertation, we assume the data space $\mathbb{E}$ to be a d-dimensional Euclidean space $\mathbb{R}^d$. We are given a set $\mathcal{O}$ of n objects. Each *object* $o \in \mathcal{O}$ has d real-valued attributes and is modeled as a point $(v_1, \ldots, v_d) \in \mathbb{E}$. For example, one way to index text documents is based on the vector space model of information retrieval. In this model, each attribute represents an index term (resp. a concept if statistical model such as probabilistic latent semantic analysis (PLSA) is applied to the collection of texts) and the space is referred to as a term space (resp. semantic space). Each document (e.g., web page, news, blogs) is represented as a point $(v_1, v_2, \cdots, v_d)$ in the multi-dimensional term space (resp. semantic space). In the simplest case, v_i is the number of occurence of term i in the document (resp. weight

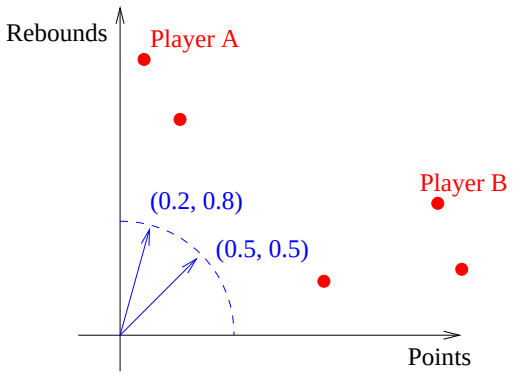


FIGURE 1.1: Example of a preference query. Player A ranks first w.r.t. a user's preference $(w_1, w_2) = (0.2, 0.8)$; player B ranks first w.r.t. a user's preference $(0.5, 0.5)$.

of concept i in the document). In practice, the coordinate v_i is rescaled based on the importance of the document, inverse document frequency of term i, etc.

Events are modeled as modifications (insertions, deletions, or updates) to the database. User queries can be *static* or *continuous*. Each snapshot query is a pair $\langle \star, k \rangle$, where $\star$ is either a user preference q or an object o. For continuous queries, users express their interests in terms of subscriptions. Let $\mathcal{S}$ denote the set of subscriptions. Each subscription $s \in \mathcal{S}$ is a triple $\langle \sigma, q, k \rangle$, where $\sigma \subset \mathbb{E}$ is the data of interest, q is a user preference, and k is the number of top-ranked objects (w.r.t. preference q) to track. Specifically, we consider the following user queries:

Preference top-k query $\langle q, k \rangle$. A natural way of ranking objects is with an object scoring function whose parameters are set according to a user's preference. A simple but effective scoring function q is a linear combination of the attribute values, where the weight associated with each attribute reflects the users' interest in this attribute. For example, a NBA fan may choose a scoring function

$$w_1 \times \#\text{ points} + w_2 \times \#\text{ rebounds} + w_3 \times \#\text{ assists}$$

for tracking all-around players, where w_1, w_2, and w_3 are the attribute weights; see Figure 1.1. As another example, an investor in a stock market may prioritize the

stock with

$$w_1 \times \text{max. swing during last 30 min} + w_2 \times \text{52w price change}.$$

Consider again the example of document retrieval. The score of a document w.r.t. a query is a variant of the cosine similarity between the document d and the query s. For example, the scoring function in the open-source Lucene [2] search engine is:

$$\sum_i s_i \times idf(i)^2 \times \sqrt{d_i} \times c_i,$$

where d_i (resp. s_i) is the frequency of term i in document d (resp. query s), $idf(i)$ is the inverse document frequency of term i, c_i is a constant depending on 1) the length and importance of document d and 2) the importance of term i. This scoring function is an instance of preference top-k query, where the i-th coordinate of an object is $idf^2(i) \times \sqrt{d_i} \times c_i$ and the i-th attribute weight is $s_i / \sqrt{\sum_i s_i^2}$.

Reverse top-k query $\langle o, k \rangle$. It was introduced by Vlachou et al. [115]. Here, each $s \in \mathcal{S}$ has a different user preference q. For a new object $o \notin \mathcal{O}$, we want to find which subscription q would rank the new object in their top k. Reverse top-k queries have applications market research [115] (e.g., what-if analysis of how much interest a new product will generate).

Continuous preference top-k query $\langle \mathbb{E}, q, k \rangle$. The problem of scalably processing a large number of continuous top-k queries [85] can be thought of as a fully dynamic version of the reverse top-k query processing problem. Again, each $s \in \mathcal{S}$ has a different user preference q. In addition to handling changes to the set $\mathcal{S}$ of subscriptions, we need to maintain the top k objects for $\mathcal{S}$ when objects are inserted, deleted, or updated.

[2] lucene.apache.org

Consider again the example of portfolio monitoring. An investor in a stock market needs to monitor the stock market in real time to identify profitable trades. Her top-k list of stocks must be maintained as the market moves. In markets such as stocks, futures, and online auctions, both the volume of object updates and the number of preferences can be large, and the processing time requirement is demanding.

Continuous range top-k query $\langle \sigma, q^*, k \rangle$. Consider a range top-$k$ query over a database of objects (e.g. stocks). The query examines a subset of the objects satisfying a range condition (e.g., stocks with risk rating between medium high and high), and picks the top k objects within this subset by a scoring function q^* (e.g., stocks with the k lowest price-to-earning ratios). Here, we consider the same scoring function q^* for every subscription $s \in \mathcal{S}$, but q^* needs not be linear. Similar to continuous preference top-k query, when the set of objects or their attribute values change over time, we keep the top-k objects of the subscriptions up to date.

1.3 Contributions

This dissertation presents geometric frameworks and scalable algorithms for answering the top-k queries described above. By mapping data and users to another geometric space, the set of affected subscriptions can be clustered more effectively; they can be described using basic geometric shapes. Consequently, notification messages (descriptions of the affected subscriptions) can be effectively compressed, and traffic can be reduced in a content-driven network. This dissertation also presents solution for designing an efficient content-driven network by joint optimizating event processing and notification dissemination. For more details, below is a summary of each main chapter:

Scalable continuous query processing under user preferences. Chapter 3 presents a scalable solution for reverse top-k queries and continuous preference top-k queries, through

the use of geometric methods. This is the first work that a reverse top-k query can be answered in sublinear time using linear-size index given fixed dimensionality d. For low dimensions ($d \leq 3$), the query time is $O(\log m+k)$, which is optimal. For continuous preference top-k queries, a dynamic hybrid approach is developed to update the affected top-k lists—driving through the individual preference or through the query response surface. This chapter also defines an approximate version of the problem and present a solution significantly more efficient than the exact one with little loss in accuracy.

Supporting user preferences in high dimensions. Supporting linear preference top-k queries and reverse top-k queries are challenging for high dimensions. Existing algorithms do not scale well in the sense that either query time or space complexity is exponential in d. Chapter 4 presents an efficient algorithm based on a dimension-reduction framework for top-k and reverse top-k queries in high dimensions. It is effective when most of the preferences are sparse—i.e., each of them specifies non-zero weights for only a small number (say ~ 2—6) of attributes. They need not specify the same subset of attributes or similar weights on attributes. Experiments show that for workloads where preferences are often sparse—a case that arises naturally in practice—the algorithm offers a desirable trade-off between speed and accuracy, which makes scalable processing of top-k and reverse top-k queries in high dimensions a reality.

Processing and notifying range top-k subscriptions. Chapter 5 considers how to support a large number of users over a wide-area network whose interests are characterized by range top-k continuous queries. Given an object update, users whose top-k results are affected need to be notified. Simple solutions include using a content-driven network to notify all users whose interest ranges contain the update (ignoring top-k), or using a server to compute only the affected queries and notifying them individually. The former solution generates too much network traffic, while the latter overwhelms the server. In

this chapter, by using a geometric framework, the set of affected queries is described succinctly with messages that can be efficiently disseminated through a content-driven network. Fast algorithms are given to reformulate each update into a set of messages whose number is provably optimal, with or without knowing all user interests. This chapter also presents several extensions to the solution, including an approximate algorithm that trades off between the cost of server-side reformulation and that of user-side post-processing, as well as efficient techniques for batch updates.

Dissemination network design. Chapter 6 studies how to assign subscriptions to brokers such that the network cost is minimized. In most previous work, subscribers are assigned to brokers according to either the closest-broker strategy [13] or interest partitioning strategy [53]. Neither approach is attractive: The former may propagate every event to nearly every broker and the latter one may assign a lot of subscribers to remote brokers. Some previous work also assumes that subscribers are randomly assigned to brokers which satisfy a set of constraints [91, 92]. In contrary to previous approaches which can be classified into either event space optimization or network space optimization, this chapter shows the importance of jointly considering the correlation between the network and event spaces. If one of the spaces is neglected, the strategy will perform very poorly on some of the network metrics. Furthermore, this chapter presents a Monte Carlo algorithm and a simple greedy algorithm for finding a broker-subscriber assignment. By simultaneously capturing spatial coherence in the network space and subscription clustering in event space, the Monte Carlo algorithm returns a broker-subscriber assignment that meets a set of network performance goals. Because of its theoretical properties and robustness to workload variations, it can serve as a reasonable yardstick in evaluate other algorithms. With its help, the greedy algorithm is concluded to work well for the subscriber assignment problem.

2

Preliminaries

This chapter first introduces a few geometric concepts, which will be used in subsequent chapters. Then it summarizes the basic concepts of publish/subscribe systems.

2.1 Geometric concepts

This section introduces few geometric concepts—duality, arrangement, coreset, and range searching. Duality will be used to map the input data and queries to another geometric space, where events can be processed more efficiently. In the new geometric space, range searching will be performed for each event update to compute a set of users whose top-k lists are changed. The arrangement of hyperplanes will also be used to explore opportunities to jointly process those affected users. Finally, if approximate solution is acceptable, the coreset techniques will be applied to 1) obtain approximate answers to the top-k queries and 2) maintain the approximate top-k list for each user.

Duality. The *duality transform* (see [82] for details) maps a point $P = (p_1, \ldots, p_d) \in \mathbb{R}^d$ to the hyperplane $P^* : x_d = p_1x_1 + \ldots + p_{d-1}x_{d-1} - p_d$; and it maps a hyperplane $h : x_d = a_1x_1 + \ldots + a_{d-1}x_{d-1} + a_d$ to the point $h^* = (a_1, \ldots, a_{d-1}, -a_d)$. It can be

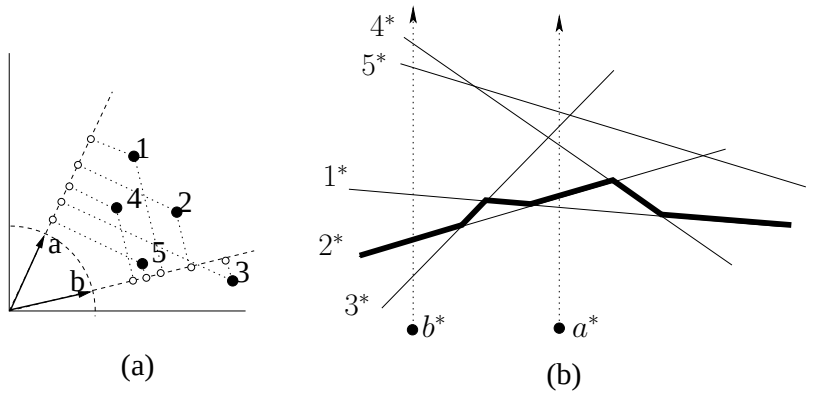
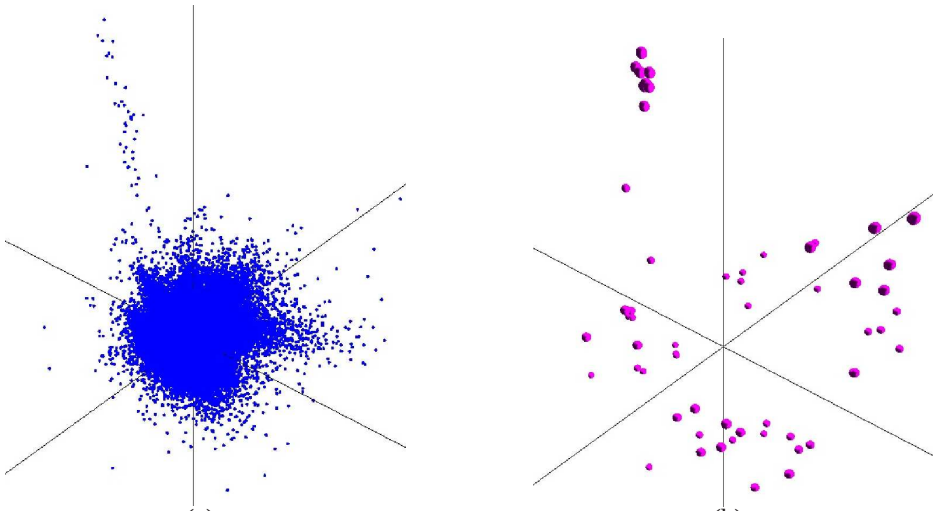


FIGURE 2.1: Duality transform. **a)** Primal space. **b)** Dual space.

verified that the dual of P^* is P itself, i.e., $P^{**} = P$, and that if P lies above (resp. below, on) h, then h^* lies above (resp. below, on) P^*. For a unit vector $w \in \mathbb{S}^{d-1}$ with $w_d \neq 0$, the set of hyperplanes normal to w, i.e., those of the form $\langle x, w \rangle = t$ where $t \in \mathbb{R}$, map to the vertical (x_d-axis parallel) ray $w^* = \{(-w_1/w_d, \ldots, -w_{d-1}/w_d, t) \mid t \in \mathbb{R}\}$; w^* is oriented in $(+x_d)$-direction (resp. $(-x_d)$-direction) if $w_d > 0$ (resp. $w_d < 0$).

Let $\mathcal{P} = \{P_i \mid 1 \leq i \leq n\}$ be the set of points. Let $\mathcal{P}^* = \{P_i^* \mid 1 \leq i \leq n\}$ be the set of hyperplanes dual to the points in $\mathcal{P}$. For a unit vector w, if $\langle P_i, w \rangle > \langle P_j, w \rangle$, then ray w^* intersects P_i before P_j. Figure 2.1 illustrates this concept. In the primal space, $\langle 1, a \rangle > \langle 2, a \rangle > \langle 4, a \rangle$ and $\langle 3, b \rangle > \langle 2, b \rangle > \langle 1, b \rangle$. In the dual space, hyperplanes 1^*, 2^*, and 4^* (resp. 3^*, 2^*, and 1^*) are the first three hyperplanes intersected by the ray a^* (resp. b^*).

Arrangement. Let H be a set of hyperplanes in $\mathbb{R}^d$. The *arrangement* of H, denoted by $\mathcal{A}(H)$, is the decomposition of $\mathbb{R}^d$ into *faces* induced by H, such that each face is the maximal connected region of $\mathbb{R}^d$ that lies in the same subset of H. $\mathcal{A}(H)$ is composed of $O(|H|^d)$ i-dimensional faces for $i = 0, \ldots, d$. See [11] for details. The *level* of a point p with respect to H, denoted by $\lambda(p, H)$, is the number of hyperplanes of H lying on or below p. Note that all points lying on the same face of $\mathcal{A}(H)$ have the same level. For $1 \leq k \leq |H|$, the *k-level* of $\mathcal{A}(H)$, denoted by $\mathcal{A}_k(H)$, is the closure of facets of $\mathcal{A}(H)$



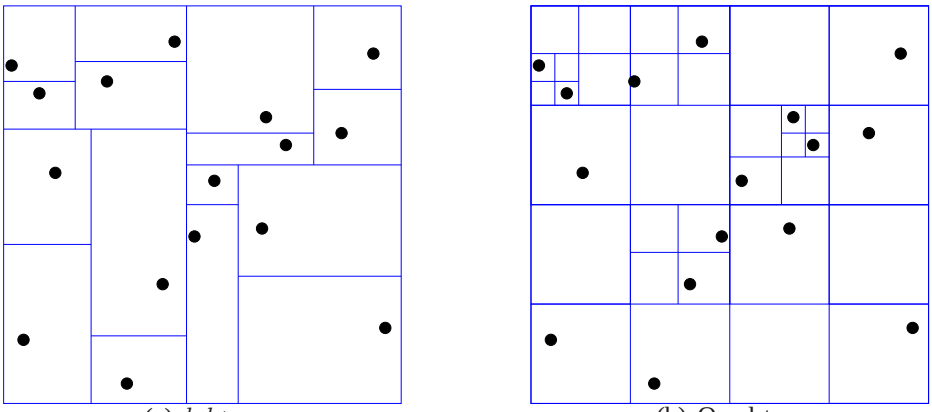
(a) (b)

FIGURE 2.2: **a)** Original point set $\mathcal{O}$; **b)** A coreset for $\mathcal{O}$ which approximates the directional width of $\mathcal{O}$.

whose level is k. $\mathcal{A}_k(H)$ is a piecewise-linear surface, and any line parallel to the x_d-axis intersects $\mathcal{A}_k(H)$ once; see Figure 2.1(b). Arrangement will be used for ranking objects in the new geometric space.

Coreset. Let $\mathcal{O}$ be a set of data objects. For many geometric problems, there exists a small *coreset* $\mathcal{C} \subset \mathcal{O}$, such that the optimal solution for $\mathcal{C}$ is an $(1+\epsilon)$-approximate solution to the original set $\mathcal{O}$, i.e., $\|f(\mathrm{OPT}_{\mathcal{O}}) - f(\mathrm{OPT}_{\mathcal{C}})\| \leq \epsilon f(\mathrm{OPT}_{\mathcal{O}})$, where f is an objective function and $\epsilon > 0$. An example of $\mathcal{C}$ is shown in Figure 2.2. One important property of $\mathcal{C}$ is that its size does not depend on the number of objects in the system; it depends on ϵ. Therefore, even if $|\mathcal{O}|$ continues to increase at an astounding rate, the coreset size still remains small. To compute an approximate solution, the coreset $\mathcal{C}$ is first computed and then taken as input to the algorithm, which runs fast due to small input size. Thus, the coreset technique can often be used to process a large-scale data efficiently.

The coreset technique will be used for different geometric problems in different chapters. Chapters 3 and 4 involve the computation of directional width; the directional width of a point set $\mathcal{O}$ w.r.t. direction $w \in \mathbb{S}^{d-1}$ is defined as $\max_{o\in\mathcal{O}}\langle w, o\rangle - \min_{o\in\mathcal{O}}\langle w, o\rangle$. The



(a) kd-tree (b) Quad-tree

FIGURE 2.3: Data structures for range searching.

directional width of coreset $\mathcal{C} \subset \mathcal{O}$ is guaranteed to approximate the directional width of $\mathcal{O}$ within a factor of $(1+\epsilon)$. Chapter 6 will solve a variant of the set covering problem: Given a set $\mathcal{O}$ of hyper-rectangles and a set P of points, we choose a subset of $\mathcal{O}$ to cover all points in P such that the sum of volumes of the rectangles in the cover is minimized subject to a set of constraints. If we choose rectangles from $\mathcal{C}$ instead of $\mathcal{O}$, the sum of volumes of the rectangles in the cover is still guaranteed to be within $(1+\epsilon)$ of the optimal solution. More details will be provided later in those sections.

Range searching. For the range searching problem, a set $\mathcal{O}$ of n objects (e.g. points, rectangles, polygons, etc.) is preprocessed such that the objects of $\mathcal{O}$ lying inside a query range (e.g. halfspace, axis-aligned rectangles, simplices, discs, etc.) can be reported or counted efficiently. Although there can be 2^n possible subsets of $\mathcal{O}$, possible answers usually consist of only a small fraction of those subsets. For example, the number of possible answers to 2-dimensional axis-aligned rectangles is n^4. Depending on the object and range types, different data structures have been developed for efficient range queries; see survey [4].

Let $\mathcal{O}$ be a set of points. Practical data structures that work for a broad range of queries are trees based on some hierarchical spatial partitioning scheme, such as a kd-tree or quad-

tree; see Figure 2.3. A kd-tree is a binary search tree which stores $O(1)$ points of $\mathcal{O}$ at each leaf. Each internal node v is split by a hyperplane perpendicular to one of the d-dimensions (which may simply be chosen in round robin fashion). Suppose x-axis is chosen as the split dimension and $\mathcal{O}_v$ is the set of points at node v. All the points in $\mathcal{O}_v$ with x-coordinate less than the median x-coordinate of $\mathcal{O}_v$ are on one side of the splitting hyperplane and the remaining points are on the other side. A quad-tree, on the other hand, decomposes each internal node v into 2^d children of equal size. Let B_v be the bounding box at node v; B_{root} contains $\mathcal{O}$. For each child of v, the side length of its bounding box is exactly half the side length of B_v. The space complexity can be reduced by compressing nodes with a single child.

A range query can be answered using a kd-tree or quad-tree in a straightforward top-down manner: Given a query range R, the tree is searched top-down as follows. At a node v with bounding box B_v, if B_v does not intersect R, the search algorithm does nothing; otherwise, the algorithm recursively searches all children of v, or, if v is leaf, return all points indexed by v that lie inside R. In the worst case, kd-tree answers a d-dimensional range query in time $O(n^{1-1/d} + t)$ using $O(dn)$ space, where t is the output size.

Since this dissertation is not focused on developing the best possible index for range searching, a kd-tree is implemented for a static set of points; a quad-tree, which avoids the balancing issue, is implemented for a dynamic set of points.

2.2 Publish/Subscribe Systems

Publish/subscribe is a model of data dissemination, where *publishers* (data providers) selectively and aperiodically push *events* to *subscribers* (users) according to their specified interests. A publish/subscribe system typically consists of an overlay network of *brokers* (servers). Publishers and subscribers are assigned to different brokers who are responsible for routing events between publishers and subscribers.

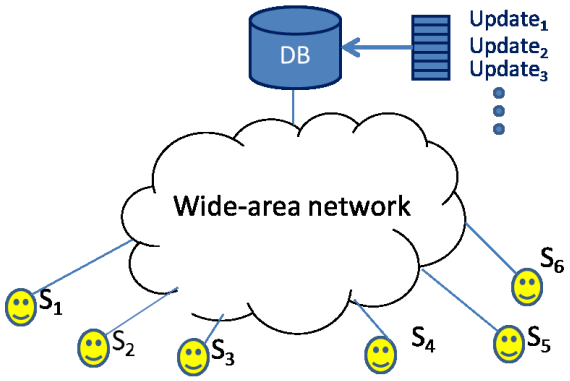


FIGURE 2.4: Publish/Subscribe.

The publish/subscribe model decouples publishers and subscribers in both time and space. Publishers need not know the locations and interests of their subscribers, who can remain anonymous to each other. On the other hand, subscribers are not required to know the identity of publishers and the time of event notifications. Because of the decoupled communication between publishers and subscribers, the publish/subscribe model is better suited for ensuring scalability, flexibility, and manageability.

Topic-based vs. content-based. Traditionally, publish/subscribe systems are *topic-based* [89, 95], in which subscribers can subscribe only to a set of predefined topics. Each event is tagged with one or more topic names and it is disseminated to all subscribers who have subscribed to those topics by using routing table lookups. However, the topics are very often too coarse-grained to fit the interests of subscribers at the individual level. In addition, if an event matches multiple topics, multiple copies of the event may have to be sent over the same link, potentially causing link congestions. The growing need for heterogeneity and expressiveness to avoid propagating excessive events has led to the growing interests in *content based* publish/subscribe systems, in which events are not constrained to belong to a specific topic. Instead, subscribers specify their interests as filters over the event contents, and routing is based on the data being transmitted instead of specifying destinations in notification messages. The two most popular content-based semantics are

predicate-based [34] and XML-based [52, 53]. For the predicate-based semantics, each event contains a list of attribute-value pairs and each subscription is a Boolean predicate against arbitrary attributes in an event. For instance, attributes in a stock update may include *Symbol*, *Price*, etc. A stock subscription may be *(Symbol = "MSFT", Price* $\geq$ *35)*. For the XML-based semantics, events are constructed as XML documents, and subscriptions are defined as XPath filters [16, 94] or other variants on individual XML documents. The additional flexibility of content-based semantics on expressiveness comes at the expense of burdening the underlying system to perform subscription matching.

Network topologies. In the publish/subscribe model, brokers communicate with each other to coorperatively distribute the subscription matching and event delivery tasks across a wide-area network. The popular interconnection topologies include *star*, *hierarchical tree*, and *general graph*. Star is a centralized server topology, which assumes there exists one single broker between publishers and subscribers. Hierarchical tree is a straightfoward extension of a star topology. For content-based systems, a parent broker only forwards an received event to the subtrees which contain matching subscriptions, therefore, unnecessary traffic would be filtered out by ancestral brokers and would never reach the low level of the hierarchical tree. The drawback of this topology is that brokers high in the hierarchical tree may be potentially overloaded. Furthermore, since there is only one single path between every pair of brokers, every broker is a critical point of failure for the entire network. On the other hand, general graph provides redundancy in the topology as well as more flexibility for configuring the broker network. Its drawback is the need to avoid cycles and choose the best paths.

Event dissemination. The common dissemination mechanisms for publish/subscribe systems include *unicast*, *broadcast*, *multicast*, and *content-based networking*. A straightforward way for a server to notify a set of matching subscriptions is to unicast each of

them individually. Alternatively, an event can be broadcast to the entire network whenever a publisher issues the event. If the event matches a subscription, the subscriber will be locally notified by its broker representative. These two approaches have their own disadvantages: For unicast, when the number of matching subscriptions is huge, the server may be overwhelmed by the outgoing traffic. For broadcast, if subscribers who share common interests are clustered together in the network, a lot of network traffic is unnecessary. On the other hand, multicast provides a good interface for topic-based publish/subscribe services. For publish/subscribe systems, multicast is more often supported at the application level by using an overlay network [36, 100, 125], which implements a distributed hash table (DHT) interface for addressing data in the network [101, 110].

For content-based publish/subscribe systems, a content-driven network [13, 35, 39] is usually used to support filter subscriptions. In the content-driven network, each destination is specified as a set of predicates, so the flow of an event is driven by the content of the event. Event is disseminated in a multi-hop manner over an overlay network of brokers. Every broker maintains a forwarding table which stores predicates indicating the conditions under which an event needs to be forwarded to a particular broker neighbor. When an event arrives at a broker, the broker 1) determines the set of next-hop destinations by matching the content of the event against the set of predicates in the forwarding table and 2) updates the forward table. The network is responsible for disseminating every event to all the nodes that have predicates matching the event. Many such systems have been built using DHTs [1, 61, 105].

3

Continuous Preference Top-k Queries

This chapter addresses the problem of scalably processing a large number of continuous preference top-k queries, through the use of geometric methods. It develops a dynamic index for supporting the *reverse top-k query*, which is of independent interest. Combining this index with another one for top-k queries, a scalable solution for processing many continuous preference top-k queries is developed by exploiting the clusteredness in user preferences. This chapter also defines an approximate version of the problem and presents a solution significantly more efficient than the exact one with little loss in accuracy.

3.1 Introduction

In many applications, users are interested only in a small number (say, k) of "top" objects from a large set. If the objects have multiple numeric attributes, how to rank these objects depends on each user's preference, oftentimes specified as vector of weights that defines a linear combination of the attribute values. The weight associated with an attribute reflects the "importance" of that attribute to the user. For example, a real estate agency may list houses for sale with attributes such as listing price, year built, size of living area, lot size,

etc. Each user is shown the highest ranked houses according his or her preference, i.e., those with the highest results for the linear combination. A user who cares most about the size of living area may assign the largest weight to this attribute (assuming that values of different attributes have been appropriately normalized relative to each other). On the other hand, a user who enjoys a yard more than indoor space may give the lot size a larger weight than the size of the living area. Because of the wide range of applications, there has been a lot of work on preference top-k queries [44, 49, 50, 67, 113].

Motivated by applications in business analysis, Vlachou et al. introduced the "reverse" top-k query [115]. In this setting, a set of user preferences is given in addition to the set of objects of interest. For a new object, the goal is to find which users would rank the new object in their top k; this information would allow a business analyst to assess, for example, the impact of a new product (object) on customers (users) relative to existing products.

Beyond the reverse top-k query, application settings such as data stream monitoring and publish/subscribe give rise to the problem of scalably processing a large number of *continuous* top-k queries [85], which can be thought of as a fully dynamic version of the reverse top-k query processing problem. In addition to handling changes to the set of user preferences, a list of top-k objects is maintained under each user preference when objects are inserted, deleted, or updated. Consider again the example of real estate listing. Houses may come on and go off the market, and their information may be updated (such as lowering the listing price); users need to be notified of the changes (if any) to their top k houses. A new or updated house may make its way into some user's top-k list, while a deleted or updated listing may remove a house from a top-k list—in which case a replacement k-th ranked house must be added to the list. As another example, consider an investor who monitors the stock market in real time to identify profitable trades. The stocks will be ranked according to a wide range of numeric attributes, including, for example, trade volume and price change since market opening today, maximum swing during the last 30

minutes, price-to-earning ratio, average analyst rating, etc. Ranking preferences depends on whether the list identifies buying or selling opportunities, and will vary according to one's personal investing style and tolerance for risk. Each top-k list must be maintained as the market moves. In markets such as stocks, futures, and online auctions, both the volume of object updates and the number of preferences can be large, and the processing time requirement is demanding.

Despite much related work under various settings, e.g., [49, 85, 115], there still lacks a scalable, comprehensive solution to the problem of processing a large number of continuous top-k queries. Earlier results [49, 85, 115] rely heavily on heuristics, which have worked for the problem sizes they were intended for. However, they have linear query time or quadratic space in the worst case, unable to handle dynamic updates efficiently, and are difficult to scale up further. For example, Mouratidis et al. [85] capped evaluation at 5,000 preferences; Vlachou et al. [115] tested up to 150,000 preferences, but the workloads did not include object updates, which are expensive under their approach. This chapter aims to scale to a million preferences with both object and preference updates.

Approach and contributions. This chapter approaches the problem of processing a large number of continuous top-k queries with a geometric framework. Preference top-k queries are closely related to the concepts of *arrangement* and k*-level* [11] in discrete geometry, as previous work on ad hoc top-k queries by Das et al. [49] has identified. This chapter offers an intuitive interpretation of the k-level as a *query response surface (QRS)*, which geometrically represents the k-th ranked object over the space of all possible preference vectors. Within this framework, three novel ideas are applied in the setting of scalable continuous top-k query processing:

- ***Connection to halfspace range queries:*** This chapter draws the connection between *halfspace range queries* [2, 8, 38, 45] and reverse top-k queries. This connection allows us to leverage results in computational geometry on halfspace range searching

to devise an index for reverse top-k queries, which, in addition to being of independent interest, serves as a critical component of the solution to the scalable continuous top-k query processing problem.

- ***Combining preference- and QRS-driven processing:*** Sometimes multiple preference top-k queries need to be evaluated simultaneously. Specifically, deleting an object may necessitate computing the new k-th ranked object for many preferences. A *preference-driven* approach runs these queries independently, which is suboptimal for clusters of preferences that share common top-k results. A *QRS-driven* approach identifies regions of the QRS within which top-k queries return the same k-th ranked object, and evaluates a single query for all preferences in each such region. However, the QRS, which depends only on the object distribution, can become very complex in high dimensions, with many regions containing few or no preferences at all. This chapter proposes a hybrid approach that combines the best of both approaches—using preference-driven processing for regions with few or no preferences, and using QRS-driven processing for dense clusters of preferences.

- ***Approximation:*** Not all applications require exact answers. By approximating QRS with a simpler surface, this chapter reduces its complexity and, in turn, improves the efficiency of the algorithms to be presented in this chapter. Specifically, the notion of *coresets*, which has been successfully used for geometric approximation algorithms [5, 6], is used to maintain a small subset of objects that induce a QRS closely approximating the QRS induced by the entire set of objects. Surprisingly, the size of the subset depends only on k and the approximation error, and not on the number of objects.

The framework and ideas to be presented in this chapter lead to the following results:

- Leveraging the connection between reverse top-k and halfspace range queries, data

structures for reverse top-k queries are obtained with linear space and sublinear query time in any fixed dimension. Experiments show orders-of-magnitude performance improvement and better scalability over the previous solution [115].

- A scalable, comprehensive solution is provided for processing a large number of continuous top-k queries. Our solution is fully dynamic in that it handles both object and preference updates efficiently. Experiments show that the hybrid approach achieves good performance by exploiting the clusteredness in user preferences while avoiding maintaining the full QRS.

- This chapter defines and solves a novel, approximate version of the problem. Experiments show that approximation significantly reduces processing costs with little loss in accuracy, allowing the solution to scale to even larger problem sizes.

As we shall see in Section 3.7, our framework and solutions can apply to settings beyond those targeted in this chapter, such as reverse nearest-neighbor queries, and preferences that are unknown or uncertain.

3.2 Preliminaries

3.2.1 Problem Statement

An *object* has d real-valued attributes and is represented as a point $(v_1, \ldots, v_d) \in \mathbb{R}^d$. A *preference* is represented as a unit vector, i.e., a point $(w_1, \ldots, w_d)$ on $\mathbb{S}^{d-1}$, the $(d-1)$-dimensional unit sphere embedded in $\mathbb{R}^d$. Each $w_i \geq 0$ is the *weight* for the i-th attribute. The *score* of an object o with respect to a preference q is $\langle q, o \rangle = \sum_{1 \leq i \leq d} w_i v_i$. A hyperplane h normal to a preference vector q is of the form $\langle q, x \rangle = t$ for some $t \in \mathbb{R}$. All objects lying on h have the same score with respect to q, namely t.

Let $\mathcal{O} = \{o_1, o_2, \ldots, o_n\} \subset \mathbb{R}^d$ denote the set of n objects of interest. For simplicity, assume that no two objects have the same score for any preference considered. With a

slight care, our framework and algorithms can be extended to handle ties. For a preference q, let $\pi_i(q, \mathcal{O})$ denote the *i-th ranked object* in $\mathcal{O}$ with respect to q; i.e., there are exactly $i-1$ objects $o' \in \mathcal{O}$ with $\langle q, o' \rangle < \langle q, o \rangle$. Let $\pi_{\leq i}(q, \mathcal{O}) = \{\pi_j(q, \mathcal{O}) \mid 1 \leq j \leq i\}$ denote the top i objects in $\mathcal{O}$ with respect to q. Geometrically, if the objects of $\mathcal{O}$ are projected onto a line parallel to q, then $\pi_i(q, \mathcal{O})$ is the i-th farthest object on this line. Alternatively, if a hyperplane normal to q is swept from $+\infty$ to $-\infty$, i.e., varying t from $+\infty$ to $-\infty$ for a hyperplane of the form $\langle q, x \rangle = t$, then $\pi_i(q, \mathcal{O})$ is the i-th object met by this hyperplane. For example, in Figure 2.1(a), $\pi_{\leq 5}(a, \mathcal{O}) = \langle 1, 2, 4, 3, 5 \rangle$; $\pi_{\leq 5}(b, \mathcal{O}) = \langle 3, 2, 1, 5, 4 \rangle$. The following two queries are subjects of interest:

- ***(Preference) top-k query:*** Given a query preference q, return $\pi_{\leq k}(q, \mathcal{O})$.
- ***Reverse (preference) top-k query:*** Given a set of m preferences $\mathcal{Q} = \{q_1, q_2, \ldots, q_m\}$ and a query object o, find the subset $\mathcal{Q}_o = \{q \in \mathcal{Q} \mid o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})\}$, i.e., all preferences in $\mathcal{Q}$ for which o is one of the top-k objects.

In the fully dynamic version of the problem, called ***scalable continuous (preference) top-k query processing***,[1] given a set $\mathcal{O}$ of n objects and a set $\mathcal{Q}$ of m preferences, the top-k objects, $\pi_{\leq k}(q, \mathcal{O})$, are maintained for all $q \in \mathcal{Q}$ at all times under the following operations.[2]

- *Object insertion.* Given a new object o, find the subset $\mathcal{Q}_o = \{q \in \mathcal{Q} \mid o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})\}$ and add o to $\mathcal{O}$.
- *Object deletion.* Given an object $o \in \mathcal{O}$ to be deleted, find the subset of preferences q such that $o \in \pi_{\leq k}(q, \mathcal{O})$, compute $\pi_k(q, \mathcal{O} \setminus \{o\})$ for each such q, and remove o from $\mathcal{O}$.

[1] "Scalable" highlights the emphasis on simultaneously processing a large number of preferences given as $\mathcal{Q}$. Contrast this problem to simply "continuous query processing," which considers one continuous query.

[2] In other words, a user with preference q will be able to maintain $\pi_{\leq k}(q, \mathcal{O})$ incrementally given the output computed by our solution for the following operations. In fact, the solution by itself does not need to store this list for every preference.

- *Preference insertion.* Add a preference q to $\mathcal{Q}$. Find $\pi_{\leq k}(q, \mathcal{O})$.

- *Preference deletion.* Remove a preference q from $\mathcal{Q}$.

Note that updates of existing objects and preferences can be modeled as deletions followed by insertions. It is not difficult to extend the framework and algorithms to handle updates directly, which would be more efficient than handling them as separate deletions and insertions.

In some cases, users do not need to know the exact top k objects, as long as they see a list sufficiently "close" to the exact one. In ***continuous approximate (preference) top-k query processing***, given a user-specified error tolerance $\varepsilon \in (0, 1)$, each user with preference q maintains a set $\tilde{\pi}_{\leq k}(q, \mathcal{O})$ of k objects such that, at all times, for all $o \in \tilde{\pi}_{\leq k}(q, \mathcal{O})$, $\langle q, o\rangle \geq \langle q, \pi_k(q, \mathcal{O})\rangle - \varepsilon\bar{d}(q, \mathcal{O})$, where $\bar{d}(q, \mathcal{O}) = \max_{o\in\mathcal{O}}\langle q, o\rangle - \min_{o\in\mathcal{O}}\langle q, o\rangle$ denotes the *extent* of the set of objects along the preference vector, i.e., the difference between the maximum and minimum scores.[3] Intuitively, all objects in approximate result are guaranteed to score higher than or not far from the actual k-th ranked object.

3.2.2 Duality and QRS

This section presents the duality transform and introduces the notion of a *query response surface*, which will be useful to our algorithms.

Duality. This chapter applies the duality transform (see Chapter 2) on both $\mathcal{O}$ and $\mathcal{S}$. Let $\mathcal{O}^* = \{o_i^* \mid 1 \leq i \leq n\}$ be the set of hyperplanes dual to the objects in $\mathcal{O}$. Let $\mathcal{Q}^* = \{q_i^* \mid 1 \leq i \leq m\}$ be the set of vertical rays dual to the preferences in $\mathcal{Q}$. For a preference q, if $o = \pi_i(q, \mathcal{O})$, then o^* is the i-th hyperplane in $\mathcal{O}^*$ intersected by the ray q^*. Hence, the first i hyperplanes of $\mathcal{O}^*$ intersected by q^* are dual to the objects in $\pi_{\leq i}(q, \mathcal{O})$; see Figure 2.1(b).

[3] $\varepsilon\bar{d}(q, \mathcal{O})$ is used instead of $\varepsilon\langle q, \pi_1(q, O)\rangle$ as the error measure because the former is independent of the choice of origin and is smaller than the latter if all object attributes have non-negative values. See the last remark in Section 3.4.1 for more discussion on an alternative formulation.

Top-k query response surface. The following lemma establishes the connection between the concept of k-level and top-k queries.

Lemma 1. *For a preference q, let w_d denote its weight for the last attribute. In the case of $w_d > 0$, if the intersection point of its dual ray q^* with $\mathcal{A}_i(\mathcal{O}^*)$ [4] lies on the hyperplane o^*, then $o = \pi_i(q, \mathcal{O})$. In the case of $w_d < 0$, if the intersection point of its dual ray q^* with $\mathcal{A}_{n-i+1}(\mathcal{O}^*)$ lies on the hyperplane o^*, then $o = \pi_i(q, \mathcal{O})$.*

Hence, $\mathcal{A}_i(\mathcal{O}^*)$ encodes, for any preference with $w_d > 0$, the identity of its i-th ranked object; each facet of $\mathcal{A}_i(\mathcal{O}^*)$ corresponds to the set of preferences with $w_d > 0$ sharing the same i-th ranked object. Similarly, $\mathcal{A}_{n-i+1}(\mathcal{O}^*)$ encodes, for any preference with $w_d < 0$, the identity of its i-th ranked object. Therefore, $\mathcal{A}_i(\mathcal{O}^*)$ and $\mathcal{A}_{n-i+1}(\mathcal{O}^*)$ are viewed as the *query response surface (QRS)* for the query returning the i-th ranked object under a preference. Overall, $\bigcup_{i\in[1,k]\cup[n-k+1,n]} \mathcal{A}_i(\mathcal{O}^*)$ encodes $\pi_{\leq k}(q, \mathcal{O})$ for any possible preference q.

3.2.3 Query Primitives

Our algorithms will use the following two primitives repeatedly.

Halfspace range query. The problem is to preprocess a set P of n points in $\mathbb{R}^d$ so that all points in P lying above a query hyperplane h can be reported quickly. In dual, this problem corresponds to reporting all hyperplanes of P^* lying below the point h^*. Several approaches have been proposed for this query. For $d \leq 3$, a query can be answered in $O(\log n + t)$ time, where t is the output size, using $O(n)$ space [45, 2]. For $d \geq 4$, given a parameter $n \leq s \leq n^{\lceil d/2 \rceil}$, a query can be answered in $O((n/s^{1/\lceil d/2 \rceil}) \log n + t)$ time using $O(s^{1+\varepsilon})$ space for any $\varepsilon > 0$ [83]. The known lower bounds [29] suggest that these bounds are close to optimal. I/O-efficient indexing schemes for halfspace range queries were given in [3]; dynamic schemes were presented in [8, 38]; see also [44].

[4] The description of Arrangement $\mathcal{A}$ can be found in Chapter 2.

Since the focus of this chapter is not to develop the best possible index for halfspace range searching, experiments in Section 3.5 simply use a tree index on P based on some hierarchical spatial partitioning scheme, such as a quad-tree or kd-tree, and answer halfspace range queries as described in Chapter 2. This chapter assumes that a halfspace range query can be answered in $O(\mathsf{q}(n) + t)$ time, and a point can be inserted or deleted in $O(\mathsf{u}(n))$ time.

Top-k query. There is a close relationship between halfspace range queries and top-k queries. Indeed, let q be a query preference for which we wish to report $\pi_{\leq k}(q, \mathcal{O})$. Let h be a hyperplane normal to q of the form $\langle q, x\rangle = t$, where $t \in \mathbb{R}$. A halfspace range query is performed over $\mathcal{O}$ with respect to h. If it returns fewer than k objects, we decrease the value of t and try again. If it attempts to report more than k points, we stop, increase t, and then try again. Thus, by doing a binary search, we can find a value of t such that exactly k objects are reported. This procedure takes $O((\mathsf{q}(n) + k) \log n)$ time. Using the index of [83], the running time can be improved to $O(\mathsf{q}(n) + k)$. Conversely, an index for top-k queries, in which a user can specify the value of k as part of the query, can be adapted to answer halfspace range queries. This chapter will thus use $O(\mathsf{q}(n) + k)$ to denote the query time for a top-k index and $O(\mathsf{u}(n))$ to denote the update time. In the implementation, simply a quad-tree or kd-tree is used to answer top-k queries with a branch-and-bound method. It can be easily replaced by a more sophisticated one without affecting the rest of our solution.

Note that we are sometimes interested only in the k-th ranked object (instead of all top k objects). When k is small (i.e., $k \leq \mathsf{q}(n)$), simply running a top-k query and returning only the k-th object works well.

3.2.4 Summary of Results

First, this chapter shows that a reverse top-k query can be formulated as a halfspace range query and thus can be answered in $O(\mathsf{q}(m)+t)$ time, where m is the number of preferences and t is the number of them affected by the query object. To the best of our knowledge, this is the first linear-size index that can answer this query in sublinear time in any fixed dimension. For $d \leq 3$, the query time is $O(\log m + k)$, which is optimal. Experiments show that our approach is much faster than the current state of the art [115, 116].

Second, this chapter presents a scalable solution for processing many continuous top-k queries, which maintains $\pi_{\leq k}(q, \mathcal{O})$ for a set $\mathcal{Q}$ of preferences under both object and preference updates. Section 3.3.2 starts by outlining two approaches—preference-driven and QRS-driven—for finding the new k-th ranked object for each preference affected by an object update. The preference-driven approach evaluates one such query for each affected preference; the QRS-driven approach evaluates one query for each facet of $\mathcal{A}_k(\mathcal{O}^*)$ (within which all preferences have the same k-th ranked object). Section 3.3.2 adopts a hybrid approach that uses the preference-driven one for sparse preferences and the QRS-driven one for clustered preferences. Experiments show that this hybrid approach achieves good performance by exploiting the clusteredness of preferences while avoiding maintaining complex regions of the QRS with sparse preferences.

Third, Section 3.4 shows that if approximate answers as described in Section 3.2.1 are acceptable, one can compute a subset $\mathcal{C} \subseteq \mathcal{O}$ of size $O(k/\varepsilon^{(d-1/2)})$, such that $\langle q, \pi_{\leq k}(q, \mathcal{C})\rangle \geq \langle q, \pi_k(q, \mathcal{O})\rangle - \varepsilon\bar{d}(q, \mathcal{O})$ for all preference q. The set $\mathcal{C}$ can be maintained efficiently under insertion and deletion of objects. Experiments show that this approach significantly reduces the complexity of QRS and improves running time with little loss of accuracy.

Finally, our results have a number of applications beyond those focused on by this chapter; we discuss them in Section 3.7.

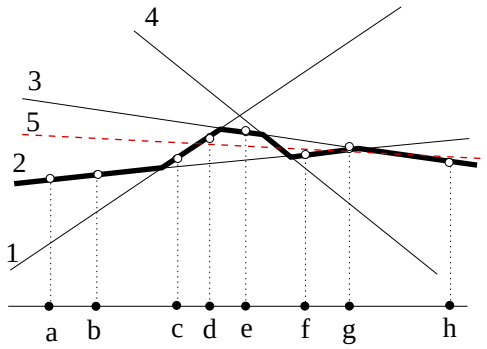


FIGURE 3.1: Illustration of object insertion in dual. Objects ($1, \ldots, 4$) are shown as solid lines and preferences ($a, b, \ldots, h$) are shown as vertical lines. Prior to inserting object 5 (shown as a hashed line), the 2-level is shown as the thick polyline, and the cutoff points are shown as white dots. Insertion of 5 changes the top-2 lists for preferences d, e, and g, whose cutoff points lie above the newly inserted dual line.

3.3 From Reverse Top-k to Continuous Top-k Queries

This section presents the solutions for answering reverse top-k queries and for processing a large number of continuous top-k queries. It starts with a static solution for reverse top-k queries, ignoring object or preference updates. It then describes a fully dynamic solution that handles both object and preference updates.

3.3.1 A Static Solution for Reverse Top-k

Given a set $\mathcal{O}$ of objects, a set $\mathcal{Q}$ of preferences, and a query object $o \notin \mathcal{O}$, we wish to report the subset of preferences $\mathcal{Q}_o = \{q \in \mathcal{Q} \mid o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})\}$. Intuitively, a preference q can be affected by o only if o scores higher than the current k-th ranked object for q. In dual, this intuition translates into the following lemma, which characterizes the set $\mathcal{Q}_o$.

Lemma 2. *For a query object o, $q \in \mathcal{Q}_o$ iff $q^* \cap \mathcal{A}_k(\mathcal{O}^*)$ lies above the dual hyperplane o^*.*

Proof. Let $p = \pi_k(q, \mathcal{O})$. By Lemma 1, $q^* \cap \mathcal{A}_k(\mathcal{O}^*) = q^* \cap p^*$. If the new object o belongs to $\pi_{\leq k}(q, \mathcal{O} \cup \{o\})$, then $p = \pi_{k+1}(q, \mathcal{O} \cup \{o\})$. By Lemma 1, q^* intersects o^* before intersecting p^*, implying that $q^* \cap \mathcal{A}_k(\mathcal{O}^*)$ lies above o^*; see Figure 3.1. □

In view of Lemma 2, the point set $\hat{\mathcal{Q}} = \{q^* \cap \mathcal{A}_k(\mathcal{O}^*) \mid q \in \mathcal{Q}\}$ is indexed in the dual space, i.e., intersection points between the vertical lines in $\mathcal{Q}^*$ and the k-level. These points are referred to as the *cutoff* points. To create this index, $\hat{\mathcal{Q}}$ is computed by performing a top-k query on $\mathcal{O}$ with each preference q to find $\pi_k(q, \mathcal{O})$; an index on $\mathcal{O}$ supporting top-k queries is described in Section 3.2.3.[5] $\hat{\mathcal{Q}}$ is then preprocessed into an index for halfspace range queries so that all points of $\hat{\mathcal{Q}}$ lying above a query hyperplane can be reported. By Lemma 2, a reverse top-k query can be answered in $O(\mathsf{q}(m) + t)$ time, where t is the output size.

Using the results in [2, 83], discussed earlier in Section 3.2.3, the following result is obtained for answering reverse top-k queries. As noted in Section 3.2.3, simpler, more practical methods can be used instead, but with weaker theoretical bounds.

Theorem 3. *Let $\mathcal{O}$ be a set of n objects in $\mathbb{R}^d$ and $\mathcal{Q}$ a set of m preferences. 1) For $d \leq 3$, $\mathcal{Q}$ can be preprocessed in $O((n+m)\log n)$ time into an index of size $O(m)$ so that a reverse top-k query can be answered in $O(\log m + t)$ time, where t is the output size. 2) For $d \geq 4$ and for a parameter $m \leq s \leq m^{\lceil d/2 \rceil}$, there is an index of size $O(s^{1+\varepsilon})$ for any $\varepsilon > 0$, so that a reverse top-k query can be answered in $O((m/s^{1/\lceil d/2 \rceil})\log m + t)$ time, where t is the output size.*

Note that the above solution allows each user (preference) to choose a different value of k; the cutoff point of each user would be defined by the value of k specific to the user.

3.3.2 A Fully Dynamic Solution

Building on the static solution for reverse top-k queries, this section shows how to process a large number of continuous top-k queries in a fully dynamic setting, with both object and preference updates. Three approaches will be discussed, and they are distinguished primarily by their handling of preferences affected by object updates. This section starts

[5] Instead of performing each top-k query individually, they can be batched using, for example, the QRS-driven or hybrid approach in Section 3.3.2.

by outlining two possible approaches with complementing strengths (and weaknesses), and then describes the hybrid approach which combines the advantages of the first two approaches. All three approaches employ an index on the set of n objects $\mathcal{O}$, which supports preference top-k queries in $O(\mathsf{q}(n) + k)$ time and object insertions and deletions in $O(\mathsf{u}(n))$ time, as discussed in Section 3.2.3.

3.3.2.1 Preference-Driven Approach

In addition to the index on $\mathcal{O}$ for top-k queries, this approach employs an index on the set $\hat{\mathcal{Q}}$ of cutoff points discussed in Section 3.3.1. Inserting a preference q involves a top-k query against the index on $\mathcal{O}$ to initialize q's list of top k objects. Then the cutoff point is computed for q from its k-th ranked object and inserted into the index on $\hat{\mathcal{Q}}$. Deleting a preference q simply entails deleting its cutoff point from the index on $\hat{\mathcal{Q}}$. Thus, the insertion and deletion times are $O(\mathsf{u}(m) + \mathsf{q}(n) + k)$ and $O(\mathsf{u}(m))$, respectively.

Now consider the insertion (or deletion) of an object o. First, this approach issues a halfspace range query with o^* against the index on $\hat{\mathcal{Q}}$ to find the set of affected preferences $\mathcal{Q}_o \subseteq \mathcal{Q}$, which correspond to the cutoff points lying above (or, for deletion of o, on or above) o^*, as in Section 3.3.1. In addition, the index on $\mathcal{O}$ is updated with o. Next, for each affected preference $q \in \mathcal{Q}_o$, this approach issues a top-k query with q against the index on $\mathcal{O}$ to find the new k-th ranked object p, and then updates the index on $\hat{\mathcal{Q}}$ with the new cutoff point for q, given by $q^* \cap p^*$. The list of top k objects for q can be easily maintained using o (or, for deletion of o, o and p). The total update time is $O(\mathsf{q}(m) + \mathsf{u}(n) + t(\mathsf{q}(n) + \mathsf{u}(m)))$, where t is the number of affected preferences.

This approach is referred to as *preference-driven* because it issues a separate top-k query for each affected subscription in $\mathcal{Q}_o$, which can be expensive if $\mathcal{Q}_o$ is large, and wasteful if many preferences share the same k-th ranked object. Intuitively, for "nearby" preferences with the same k-th ranked object, we would like to use only one query, which leads to the next approach.

3.3.2.2 QRS-Driven Approach

An alternative approach will be to leverage the query response surface $\mathcal{A}_k(\mathcal{O}^*)$. Recall from Section 3.2.2 that each facet of this QRS corresponds to a set of preferences sharing the same k-th ranked object, giving us a natural way to process preferences in groups.

To this end, in addition to the index on $\mathcal{O}$ for top-k queries, the *QRS-driven* approach maintains an index for $\mathcal{A}_k(\mathcal{O}^*)$. If an object o is inserted (or deleted), this approach updates the index on $\mathcal{O}$ as well as the index for $\mathcal{A}_k(\mathcal{O}^*)$, querying the index on $\mathcal{O}$ as needed. The complexity of this operation does not depend on the number of preferences. Finally, for each new facet φ on the updated QRS, let p denote the object whose dual hyperplane p^* contains φ, and let $\mathcal{Q}_\varphi$ denote the set of preferences q whose dual lines q^* intersect φ.[6] All preferences in $\mathcal{Q}_\varphi$ have p as their new k-th ranked object, and their lists of top k objects can be maintained using o (resp. o and p).

Note that updating of the QRS is oblivious to the actual set of preferences. Indeed, the QRS-driven approach effectively computes, without any knowledge of $\mathcal{Q}$, a description (based on facets of $\mathcal{A}_k(\mathcal{O}^*)$) of the set of affected preferences, together with the incremental changes to their lists of top-k objects. This feature makes the QRS-driven approach attractive for some applications (such as *monochromatic reverse top-k* queries in business analysis [115]), a point we shall come back to in Section 3.7.

There are two difficulties with this approach, however. First, the QRS can be large and complex to update, especially in higher dimensions. Second, many parts of the QRS may have few or no preferences, so it would be a waste of effort to maintain the QRS for these parts. These observations lead to the idea of combining this approach with the preference-driven approach earlier.

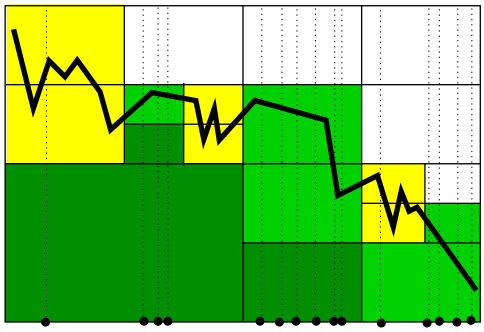


FIGURE 3.2: Leaves of $\mathcal{T}$. The QRS is shown as a thick polyline, and the dual lines of preferences are shown as dotted lines. Solid lines show the partitioning of the dual space into leaves; for clarity here equi-distance partitioning is used, though in practice it need not be the case. Black leaves are filled with a dark (green) shade; grey-dense leaves are filled with a medium (green) shade; grey-sparse leaves are filled with a light (yellow) shade; white leaves are not shaded.

3.3.2.3 Hybrid Approach

For the preference-driven approach, the index on $\hat{\mathcal{Q}}$ can be updated efficiently, but independently computing the new k-th ranked object for each affected preference can be inefficient. For the QRS-driven approach, identically affected preferences are processed efficiently as a group, but maintaining parts of the QRS with few or no actual preferences is wasteful. To get the best from both approaches, a hybrid approach is adopted to intelligently switch between the two processing modes.

In addition to the index on $\mathcal{O}$ for top-k queries, the hybrid approach maintains a search tree $\mathcal{T}$ based on a hierarchical spatial partitioning of the dual space (a quad-tree is used in the implementation). Each node v of $\mathcal{T}$ is associated with a bounding box $B_v \subseteq \mathbb{R}^d$. Let $\mathcal{Q}_v^* \subseteq \mathcal{Q}^*$ denote the set of vertical lines in $\mathcal{Q}^*$ stabbing B_v, and let $\mathcal{O}_v^* \subseteq \mathcal{O}^*$ denote the set of hyperplanes in $\mathcal{O}^*$ intersecting B_v. The following three counters are stored at each node v: $m_v = |\mathcal{Q}_v^*|$, $n_v = |\mathcal{O}_v^*|$, and b_v^Δ, the number of hyperplanes in $\mathcal{O}_{p(v)}^*$ that lie below B_v, where $p(v)$ is the parent of v. The number of hyperplanes in $\mathcal{O}^*$ lying below B_v, denoted b_v, can be computed by summing b_u^Δ over each node u on the path from the root to v. At

[6] This set can be either explicitly maintained for each facet of the QRS, or computed by searching another data structure on $\mathcal{Q}$.

each leaf v of $\mathcal{T}$, the sets $\mathcal{O}_v^*$ and $\mathcal{Q}_v^*$ are also stored.

A leaf v can be one of the following types (where τ_m and τ_n are user-defined parameters):

- *White* if $b_v > k$; i.e., B_v is strictly above $\mathcal{A}_k(\mathcal{O}^*)$.
- *Black* if $b_v + n_v < k$; i.e., B_v is strictly below $\mathcal{A}_k(\mathcal{O}^*)$.
- *Grey-sparse* if $(b_v \leq k \leq b_v + n_v) \wedge (m_v < \tau_m)$; i.e., B_v intersects $\mathcal{A}_k(\mathcal{O}^*)$ and contains few cutoff points.
- *Grey-dense* if $(b_v \leq k \leq b_v + n_v) \wedge (m_v \geq \tau_m) \wedge (n_v < \tau_n)$; i.e., B_v intersects $\mathcal{A}_k(\mathcal{O}^*)$ and likely contains many cutoff points, and $\mathcal{A}_k(\mathcal{O}^*)$ is not very complex.

These leaf types are depicted in Figure 3.2. A node v satisfying none of the conditions above passes the following *splitting condition*:

$$(b_v \leq k \leq b_v + n_v) \wedge (m_v \geq \tau_m) \wedge (n_v \geq \tau_n).$$

In this case, v is an interior node. Practically, τ_m and τ_n are chosen to reflect 1) the "tipping point" when one of the preference- and QRS-driven approaches becomes more efficient than the other, and 2) the granularity at which such a decision is made.

Constructing $\mathcal{T}$. Initially, $\mathcal{T}$ is a tree containing a single unvisited root node with $B_{\text{root}} = \mathbb{R}^d$, $m_{\text{root}} = m$, $n_{\text{root}} = n$, and $b^{\Delta}_{\text{root}} = b_{\text{root}} = 0$. The splitting condition is tested at each unvisited node v. If v passes the splitting condition, v becomes a non-leaf and B_v is partitioned among its children, each of which will be visited. Otherwise, v is a leaf: $\mathcal{Q}_v^*$ and $\mathcal{O}_v^*$ are stored, and v's type is then determined.

Object insertion. For the insertion of a new object o, $\mathcal{T}$ is first updated top-down. At a node v, n_v is incremented by 1 if o^* intersects B_v, or increment b_v^{Δ} if o^* lies below B_v. There are three cases:

1. v *was a non-leaf.* If now $b_v > k$, the subtree rooted at v is contracted into a single white leaf and stop. Otherwise, v remains a non-leaf and the same procedure is repeated for each child of v, but skipping any child u where o^* lies above B_u.

2. v *was a white or black leaf.* The only case requiring action is when a previously black v becomes grey or non-leaf because now $b_v + n_v = k$. In this case, a subtree rooted at v is constructed for $\mathcal{O}_v^*, \mathcal{Q}_v^*$ using the construction procedure above.

3. v *was a grey leaf.* The only case requiring action is when a previously grey-dense v turns into a non-leaf because n_v now reaches τ_n. In that case, the construction procedure is used to build a subtree rooted at v.

After $\mathcal{T}$ has been updated, $\mathcal{T}$ is traversed to compute, for each grey leaf v, the set of affected preferences in $\mathcal{Q}_v^*$:

- If v is grey-dense, the QRS inside B_v must be simple because few dual hyperplanes intersect B_v, so a QRS-driven approach is taken. The new facets of the QRS inside B_v (i.e., $\mathcal{A}_{k-b_v}(\mathcal{O}_v^*) \cap B_v$) is computed. For each new facet φ, let p denote the object whose dual hyperplane p^* contains φ. All preferences whose dual lines intersect φ have p as the new k-th ranked object.

- If v is grey-sparse, $\mathcal{Q}_v^*$ is small, so a preference-driven approach is taken, with one top-k query issuing against the index on $\mathcal{O}$ for each preference in $\mathcal{Q}_v^*$. If $\mathcal{O}_v^*$ happens to be small too, instead of using the index on $\mathcal{O}$, simply the $(k - b_v)$-th ranked object in $\mathcal{O}_v^*$ can be computed for each preference by scanning $\mathcal{O}_v^*$.

Object deletion. For the deletion of object o, again $\mathcal{T}$ is first updated top-down. At a node v, n_v is decremented by 1 if o^* intersects B_v, or decrement b_v^Δ if o^* lies below B_v. There are three cases:

1. v *was a non-leaf.* If $b_v + n_v$ now drops below k, the subtree rooted at v is contracted into a single black leaf. If n_v drops below τ_n but still $b_v + n_v \geq k$, the subtree rooted at v is contracted into a single grey-dense leaf. Otherwise, v remains a non-leaf and the same procedure is repeated for each child of v, but skipping any child u where o^* lies above B_u.

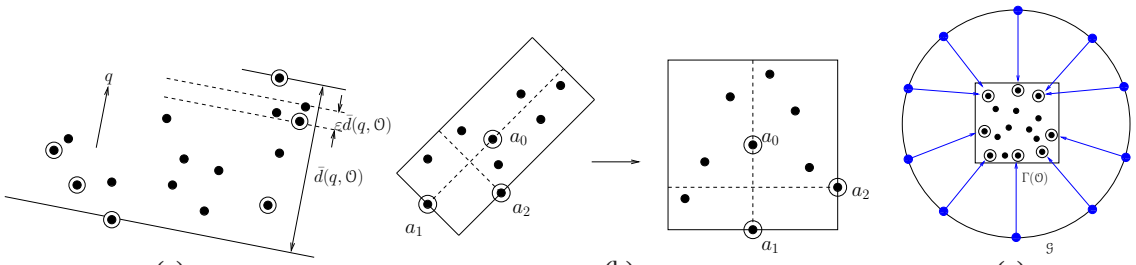
2. v *was a white or black leaf.* The only case requiring action is when a previously white v becomes grey or non-leaf because now $b_v = k$. In this case, a subtree rooted at v is constructed using the construction procedure above.

3. v *was a grey leaf.* v becomes black if $b_v + n_v < k$.

After $\mathcal{T}$ has been updated, the set of affected preferences and their new k-th ranked objects are computed. As discussed in the case of object insertion, the update algorithm switches between QRS- and preference-driven approaches as appropriate.

Preference insertion. For the insertion of a new preference q, a top-k query with q is issued against the index on $\mathcal{O}$ to find $\mathcal{O}_q = \pi_{\leq k}(q, \mathcal{O})$. Using $\mathcal{O}_q$, q's cutoff point $\hat{q}$ is calculated, and $\mathcal{T}$ is searched for the grey leaf v such that $\hat{q} \in B_v$. For every node u along the path from the root to v, m_u is incremented by 1. If v was grey-sparse and now $m_v = \tau_m$, v would become grey-dense or non-leaf; in this case, a subtree rooted at v is constructed using the construction procedure above.

Preference deletion. For the deletion of preference q, $\mathcal{T}$ is searched for the grey leaf v such that B_v contains the cutoff point of q. For every node u along the path from the root to v, m_u is decremented by 1. If (and as soon as) m_u drops from τ_m to $\tau_m - 1$ for any u encountered during the search, the subtree rooted at u is replaced with a single grey-sparse leaf.



(a) (b) (c)

FIGURE 3.3: **a)** Illustration of coreset for $k = 2$. Points are shown as black dots and members of the coreset are circled. **b)** Converting $\mathcal{O}$ into a fat point set using an affine transform defined using anchor points $\{a_0, a_1, a_2\}$. **c)** Constructing the coreset by finding the k nearest neighbors in $\Gamma(\mathcal{O})$ (showing in the bounding box) of each grid point in $\mathcal{G}$ (shown on the sphere).

3.4 Approximate Top-k Queries

As mentioned in Section 3.1, it suffices for users of many applications to have approximate lists of top-k objects under their preferences. This section shows that in this case the index can be built on a small subset of $\mathcal{O}$, and the lists can be updated more efficiently. Section 3.4.1 shows that such a small subset $\mathcal{C}$, called *coreset*, can be computed efficiently. Section 3.4.2 describes how to update the coreset efficiently as $\mathcal{O}$ changes. Section 3.4.3 further describes procedures for maintaining indexes based on $\mathcal{C}$ as well as the top-k lists of all users. As we will see, maintaining them upon every change to $\mathcal{C}$ is unnecessary and expensive—insertion or deletion of a single object in $\mathcal{O}$ sometimes causes multiple changes to $\mathcal{C}$. Therefore, these procedures are designed to perform maintenance lazily only when necessary.

3.4.1 Computing a Coreset

For a unit vector $q \in \mathbb{S}^{d-1}$, the *extent* of $\mathcal{O}$ in direction q, denoted by $\bar{d}(q, \mathcal{O})$, is

$$\bar{d}(q, \mathcal{O}) = \max_{o \in \mathcal{O}} \langle q, o \rangle - \min_{o \in \mathcal{O}} \langle q, o \rangle,$$

i.e., the difference between the maximum and the minimum scores for the preference q. Given an integer $k \geq 1$ and a parameter $\varepsilon > 0$, a subset $\mathcal{C} \subseteq \mathcal{O}$ is called a (k, ε)-*coreset*

(or simply *coreset* for brevity) if for all $i \leq k$ and $q \in \mathbb{S}^{d-1}$,

$$\langle q, \pi_i(q, \mathcal{C})\rangle \geq \langle q, \pi_i(q, \mathcal{O})\rangle - \varepsilon \bar{d}(q, \mathcal{O}). \tag{3.1}$$

This section shows that a coreset of size $O(k/\varepsilon^{(d-1/2)})$ can be computed efficiently.

Before describing the algorithm, we need a property of coreset, which will be critical for the algorithm. A linear transform $\Gamma : \mathbb{R}^d \to \mathbb{R}^d$ is called an *affine* transform if the matrix Γ is nonsingular—it includes translation, rotation, and scaling.

Lemma 4. *Let $k > 0$ be an integer, $\varepsilon > 0$ a parameter, and Γ an affine transform. A subset $\mathcal{C} \subseteq \mathcal{O}$ is a (k, ε)-coreset of $\mathcal{O}$ if and only if $\Gamma(\mathcal{C})$ is a (k, ε)-coreset of $\Gamma(\mathcal{O})$.*

The proof of this lemma, a slight variant of the one given in [122], is omitted here.

Converting $\mathcal{O}$ into a fat point set. For a constant $\alpha > 0$, $\mathcal{O}$ is called α-*fat* if

$$\max_{q_1, q_2 \in \mathbb{S}^{d-1}} \bar{d}(q_1, \mathcal{O})/\bar{d}(q_2, \mathcal{O}) \leq \alpha.$$

An affine transform Γ can be computed such that $\Gamma(o)$ is α_d-fat for some constant α_d that depends on d. To this end, the approximate minimum-volume bounding box B for $\mathcal{O}$ is first computed with the algorithm of Barequet and Har-Peled [24], as follows. The set $\mathcal{A}$ of d *anchor* objects $a_0, \ldots, a_d$ is picked, one by one. a_0 is chosen arbitrarily, and a_{i+1} is chosen to be the farthest object from $\mathrm{span}(a_0, \ldots, a_i)$, i.e., the span of all previously chosen anchors. The set $\mathcal{A}$ of anchor objects defines the bounding box B: a_0 lies in the center of B; the vector from a_{i+1} to $\mathrm{span}(a_0, \ldots, a_i)$ gives an direction orthogonal to the directions defined by $\{a_0, \ldots, a_i\}$ (see Figure 3.3(b)). Next, a transform Γ is computed, such that $\Gamma(B)$ maps to $[-1, +1]^d$. It can be checked that $\Gamma(\mathcal{O})$ is α_d-fat for a constant α_d (see, e.g., [5]).

Constructing $\mathcal{C}$. Given $\mathcal{O}$, the affine transform Γ is first computed, as described above, so that $\Gamma(\mathcal{O})$ is fat and $\Gamma(\mathcal{O}) \subset [-1, +1]^d$. Let S be the sphere of radius $\sqrt{d} + 1$ centered

at the origin in $\mathbb{R}^d$. A set $\mathcal{G}$ of *grid points* is constructed on S as follows. Set parameter $\delta = \beta\sqrt{\varepsilon}$ for a sufficiently small constant $0 < \beta < 1$. A set $\mathcal{G} \subset S$ of $O(1/\beta^{d-1}) = O(1/\varepsilon^{(d-1)/2})$ points is chosen, so that for any point $x \in S$ there is a grid point $p \in \mathcal{G}$ such that $\|x - p\| \leq \delta$.

Next, for each grid point $p \in \mathcal{G}$, $\sigma_k(p)$, the $(\varepsilon/2)$-approximate k nearest neighbors of p in $\Gamma(\mathcal{O})$, is computed; see Figure 3.3(c). $\mathcal{C}$ is set to $\bigcup_{p\in\mathcal{G}} \sigma_k(p)$. By adapting the methods for answering approximate nearest-neighbor queries [18], the $(\varepsilon/2)$-approximate k nearest neighbors of a query point can be computed in $O(\log n + k/\varepsilon^d)$ time. In practice, a branch-and-bound algorithm (similar to the one used for answering a top-k query) can be used.

Theorem 5. *Given a set $\mathcal{O}$ of n objects, an integer $k > 0$, and a parameter $\varepsilon > 0$, a (k, ε)-coreset of $\mathcal{O}$ of size $O(k/\varepsilon^{(d-1)/2})$ can be computed in $O(n \log n + k/\varepsilon^{3d/2})$ time.*

Proof. Since $|\mathcal{G}| = O(1/\varepsilon^{(d-1)/2})$, $|\mathcal{C}| = O(k/\varepsilon^{(d-1)/2})$; the running time of the algorithm follows from the query time of the approximate k-nearest neighbor data structure. It thus suffices to prove that $\mathcal{C}$ is a (k, ε)-coreset.

For each $q \in \mathbb{S}^{d-1}$ and for all $i \leq k$, we show that (3.1) holds. We prove this claim by induction on i. Suppose this claim holds for up to $i - 1$. Suppose $o = \pi_i(q, \mathcal{O})$. Let $x \in S$ be the intersection point of S with the ray emanating from o in direction q. Refer to Figure 3.4. Let $g \in \mathcal{G}$ be the closest grid point to $x \in S$. If o is the i-th nearest neighbor of g, then o is included in $\mathcal{C}$. Otherwise, the i-th nearest neighbor must lie in the shaded (blue) region. The error is within $\|w - z\|$, which can be shown to be less than $\|h - x\| < (\varepsilon/2)\bar{d}(q, \mathcal{O})$, provided that β is chosen sufficiently small; see [122]. Recall that we computed the $(\varepsilon/2)$-approximate k-nearest neighbors of g, so we can argue that if $\tilde{o}$ is the $(\varepsilon/2)$-approximate i-th nearest neighbor of g, then $\langle q, \tilde{o}\rangle \geq \langle q, o\rangle - \varepsilon\bar{d}(q, \mathcal{O})$. □

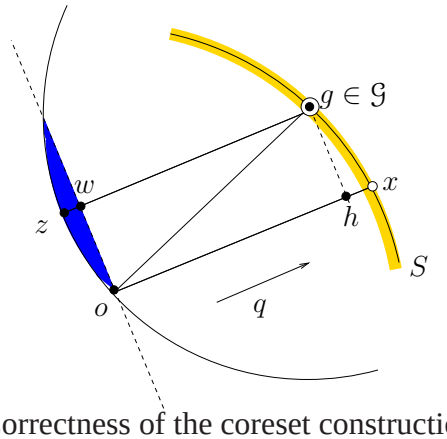


FIGURE 3.4: Correctness of the coreset construction algorithm.

Remarks. Note that $\mathcal{C}$ approximates $\pi_{\leq k}(q, \mathcal{O})$ for all $q \in \mathbb{S}^{d-1}$. If we are interested in preferences $q = (q_1, \ldots, q_d)$ for which $q_i \geq 0$ for all $i \leq d$, then we choose only those points of $\mathcal{G}$ that are within or not far from the first orthant (The implementation uses those with coordinates no less than -0.3). The asymptotic bound on the size of $\mathcal{C}$ does not change, but the constant changes.

Agarwal et al. [7] define a coreset using a stronger definition of approximation which ensures that

$$\langle q, \pi_i(q, \mathcal{C})\rangle \geq (1 - \varepsilon)\langle q, \pi_i(q, \mathcal{O})\rangle \tag{3.2}$$

for all $1 \leq i \leq k$ and for all directions $q \in \mathbb{S}^{d-1}$ if all attributes are non-negative. Using a similar algorithm they show that a coreset of size $O(k/\varepsilon^{(d-1)/2})$ can be computed under this stronger definition. A result in [9] shows that the coreset can be maintained efficiently under insertion and deletion of objects. The definition presented in this section provides a weaker theoretical guarantee because the error is bounded in terms of extent, which depends on the position of highest ranked object. In particular, if the score of $\pi_k(q, \mathcal{O})$ is much smaller than $\pi_1(q, \mathcal{O})$, then the bound in (3.1) could be large. However, the presented definition is chosen because in practice it also produces a very good approximation of $\pi_{\leq k}(q, \mathcal{O})$ for every q, and updating $\mathcal{C}$ under insertion or deletion of an object is considerably simpler.

3.4.2 Updating the Coreset

This section discusses how to maintain the coreset $\mathcal{C}$ under insertion and deletion of objects, i.e., maintain the set of anchor points $\mathcal{A}$, the bounding box B, and the affine transform Γ. To help reduce the amortized cost of reconstructing the coreset, the coreset is maintained as the union of two sets, i.e., $\mathcal{C} = \mathcal{C}^{\text{in}} \cup \mathcal{C}^{\text{out}}$, where $\mathcal{C}^{\text{out}}$ is used to "buffer" new objects that would otherwise trigger coreset reconstruction immediately; details now follow.

Object insertion. Suppose the new object o is inside the bounding box B. For each grid point $g \in \mathcal{G}$, if $\Gamma(o)$ is one of g's new (approximate) k nearest neighbors among $\Gamma(\mathcal{O} \cup \{o\})$, o is inserted into $\mathcal{C}^{\text{in}}$ and the old k-th nearest neighbor of g is removed from $\mathcal{C}^{\text{in}}$. Overall, $\mathcal{C}$ does not change unless o becomes one of the k nearest neighbors of some grid point, in which case o is inserted to $\mathcal{C}^{\text{in}}$ and one or more objects are deleted from $\mathcal{C}^{\text{in}}$.

If the new object o is outside B, the naive approach would be to reconstruct $\mathcal{C}$ because Γ needs to be recomputed. To reduce the frequency of expensive coreset reconstructions, o is simply buffered in $\mathcal{C}^{\text{out}}$, and coreset reconstruction is postponed until $|\mathcal{C}^{\text{in}}| = |\mathcal{C}^{\text{out}}|$. Immediately following a reconstruction, $\mathcal{C}^{\text{out}} = \emptyset$ and $\mathcal{C} = \mathcal{C}^{\text{in}}$.

When reconstructing the coreset, the update algorithm attempts to reuse the objects in the current coreset whenever possible. Let $\mathcal{C}'$ denote the content of $\mathcal{C}$ before reconstruction. Let $\mathcal{O}_g$ be the set of new k nearest neighbors for a grid point $g \in \mathcal{G}$. For each object $o \in \mathcal{O}_g \setminus \mathcal{C}'$, if there exists an object $o' \in \mathcal{C}' \setminus \mathcal{O}_g$, such that the distance from g to o' is approximately the same as the distance from g to o, then o is substituted with o'. This technique reduces the number of changes in the coreset membership, which in turn helps reduce the cost of maintaining the data structures built on the coreset.

Object deletion. Suppose an object $o \in \mathcal{O}$ is deleted. If $o \notin \mathcal{C}$, there is nothing to do. If $o \in \mathcal{C}^{\text{out}}$, o is simply deleted from $\mathcal{C}^{\text{out}}$. Next, suppose $o \in \mathcal{C}^{\text{in}}$.

If o is not an anchor point in $\mathcal{A}$ defining the affine transform Γ, let $\mathcal{G}_o$ denote the subset of grid points $g \in \mathcal{G}$ such that $\Gamma(o)$ is one of g's approximate k nearest neighbors. o is deleted from $\mathcal{C}^{\text{in}}$, and for each $g \in \mathcal{G}_o$, the approximate k-th nearest neighbor of g is computed and added to $\mathcal{C}^{\text{in}}$.

If o happens to be an anchor point in $\mathcal{A}$, a new affine transform is needed. Thus, the reconstruction of the coreset is triggered. Again, as discussed in the case of object insertion, the update algorithm attempts to reuse the objects in the current coreset whenever possible.

3.4.3 Updating Indexes and Top-k Lists

Recall from Section 3.3.2 that a number of indexes is maintained for scalable processing of continuous top-k queries. For example, the preference-driven approach maintains an index $\mathcal{I}$ of objects (for preference top-k queries) and an index $\mathcal{J}$ of cutoff points. The hybrid approach maintains $\mathcal{I}$ and a search tree $\mathcal{T}$. With the coreset approach, these indexes are now based on $\mathcal{C}$ instead of $\mathcal{O}$. When $\mathcal{C}$ changes, these indexes need to be updated as well as the approximate top-k lists for all preferences. Naively, they can be simply updated for every change to $\mathcal{C}$, but this strategy is expensive because a single insertion or deletion in $\mathcal{O}$ may sometimes translate to many changes to $\mathcal{C}$, as discussed in Section 3.4.2. The key observation is that it is unnecessary to carry out some updates to the indexes and top-k lists immediately. To illustrate, suppose that the insertion of an object o from $\mathcal{O}$ causes another object o' to disappear from $\mathcal{C}$. There is no need to remove o' from the top-k list of a preference, because o' has not been deleted from $\mathcal{O}$, and the old list will continue to serve correctly as an approximate top-k list. Likewise, there is no need to delete o' from the indexes.

Therefore, following this intuition, a lazy approach is used to update the indexes and the top-k lists. Two buffers are maintained:

- *Deletion buffer* stores the set ∇ of objects that have been deleted from $\mathcal{C}$ but not

from $\mathcal{O}$; these objects are still present in the index $\mathcal{I}$ of objects and the tree structure $\mathcal{T}$.

- *QRS buffer* stores a set Δ of objects that have been inserted into $\mathcal{C}$ because of other object updates (i.e., o itself was already present in $\mathcal{O}$ before it is inserted into $\mathcal{C}$); these objects are inserted into $\mathcal{I}$ and $\mathcal{T}$, but they are not used to update the index $\mathcal{J}$ of cutoff points or the top-k lists.

The ramainder of this section is devoted to describe the procedures for updating the indexes and top-k lists when an object is inserted or deleted in $\mathcal{O}$. The description covers the maintenance of $\mathcal{I}$, $\mathcal{J}$, and $\mathcal{T}$; in practice, only the subset of these indexes used by the approach chosen from Section 3.3.2 needs to be maintained.

Object insertion. Suppose a new object o is inserted into $\mathcal{O}$. Recall the coreset update algorithm in Section 3.4.2. If o does not affect $\mathcal{C}$, there is nothing to do and the algorithm stops. If o is added to $\mathcal{C}$, it is inserted into $\mathcal{I}$ and $\mathcal{T}$. The set of affected preferences is computed as discussed in Section 3.3, and update their top-k lists as well as their corresponding cutoff points in $\mathcal{J}$.

Furthermore, if the insertion of o causes a set $\mathcal{C}^-$ of objects to be removed from $\mathcal{C}$, $\mathcal{C}^-$ is inserted into the deletion buffer ∇ and avoid updating $\mathcal{I}$ and $\mathcal{T}$.

Finally, if the insertion of $\mathcal{O}$ causes a set $\mathcal{C}^+$ of objects to be added to $\mathcal{C}$ (which happens when $\mathcal{C}$ is reconstructed), each $o' \in \mathcal{C}^+$ is processed as follows. If $o' \in \nabla$, it is simply removed from ∇ and nothing further needs to be done; otherwise, o' is inserted into $\mathcal{I}$, $\mathcal{T}$, and the QRS buffer Δ, without updating $\mathcal{J}$ or any top-k lists.

Object deletion. Suppose an existing object o is deleted from $\mathcal{O}$. If o is in neither the current coreset nor the deletion buffer ∇, the algorithm simply stops. Otherwise, o is deleted from there and from $\mathcal{I}$ and $\mathcal{T}$. The set of affected preferences is also computed, and their top-k lists as well as their corresponding cutoff points in $\mathcal{J}$ are updated.

Recall the coreset update algorithm in Section 3.4.2. If o was in the $\mathcal{C}$ and the coreset is not reconstructed, then the deletion of o can cause insertion of a set $\mathcal{C}^+$ of objects into $\mathcal{C}$. As in the case of object insertion discussed above, for each $o' \in \mathcal{C}^+$, if $o' \in \nabla$, it is simply removed from ∇; otherwise, o' is inserted into $\mathcal{I}$, $\mathcal{T}$, and the QRS buffer Δ, again without updating $\mathcal{J}$ or any top-k lists.

Finally, if $\mathcal{C}$ was reconstructed as the result of deleting o, let $\mathcal{C}^+$ denote the set of objects inserted into $\mathcal{C}$ and let $\mathcal{C}^-$ denote the set of objects deleted from $\mathcal{C}$. Each object of $\mathcal{C}^-$ is inserted into the deletion buffer ∇. The processing of $\mathcal{C}^+$ is more involved. Let $\mathcal{O}' = (\mathcal{C}^+ \setminus \nabla) \cup \Delta$. The objects in $\mathcal{C}^+ \setminus \nabla$ are inserted into $\mathcal{I}$ and $\mathcal{T}$, and delete those in $\mathcal{C}^+ \cap \nabla$ from ∇. By performing a reverse top-k query for each object in $\mathcal{O}'$, the set $\mathcal{Q}_o$ of preferences that need updating is identified. Their top-k lists as well as their corresponding cutoff points in $\mathcal{J}$ are updated. Further details are omitted.

3.5 Experimental Evaluation

Approaches compared. For static reverse top-k queries, the approach based on halfspace range queries (Section 3.3.1) has been implemented using a quad-tree as the underlying index for cutoff points;[7] this algorithm is referred to as ***HSR*** for short. For comparison, the *RTOP-Grid* algorithm by Vlachou et al. [115] has been implemented, which is the most recent and most relevant to the work presented in this chapter; this algorithm is referred to as ***GRID*** for short.

For the problem of processing a large number of continuous top-k queries, all three approaches discussed in Section 3.3.2 have been implemented: preference-based, QRS-based, and hybrid. They are not compared with GRID in this case, because GRID does not handle object updates efficiently, and is already significantly outperformed by our approach in the static case (as we will see in Section 3.5.1).

[7] Experiments have also been performed with a kd-tree implementation, which showed comparable performance: it works better than the quad-tree for $d > 4$ and worse for $d < 4$. Since the choice does not change any conclusion drawn in this section, results for the kd-tree are not shown in this chapter.

For the three approaches, again quad-trees are used for the underlying indexes when applicable. For the QRS-based approach, a quad-tree is used to store the QRS, stopping when a node v's bounding box B_v is strictly above or below the QRS, or intersects fewer than τ_n hyperplanes in $\mathcal{O}^*$—analogous to the hybrid approach in Section 3.3.2.3 with $\tau_m = 0$ such that there are no grey-sparse nodes.

Finally, the coreset-based approach has been implemented for the approximate version of the problem. All algorithms are implemented in C++.

Performance metrics. The following metrics are considered when evaluating competing approaches:

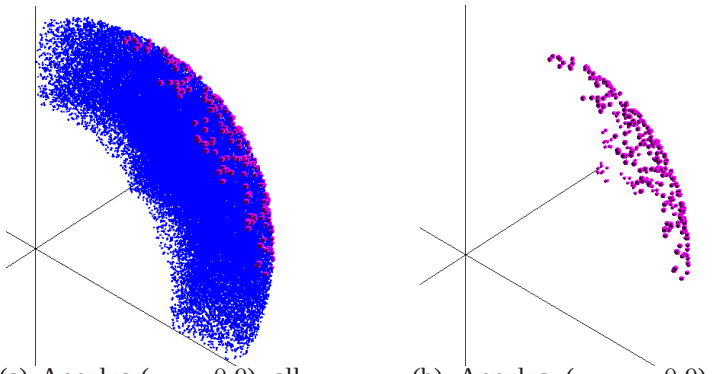
- ***Time (per request):*** The wall-clock time for handling a request, be it a reverse top-k query in the static case, or an object or preference update in the dynamic case (which includes maintenance of data structures, processing of affected preferences, etc.).

- ***# calls:*** The number of calls to query primitives—halfspace range or top-k queries—discussed in Section 3.2.3. This metric allows performance to be measured independent from particular implementations of the primitives.

- ***Approximation error (estimated):*** The relative error observed in the answers produced by the coreset-based approximation approach in Section 3.4. For a coreset $\mathcal{C} \subseteq \mathcal{O}$, the error in the top-$k$ answer for preference q is measured as

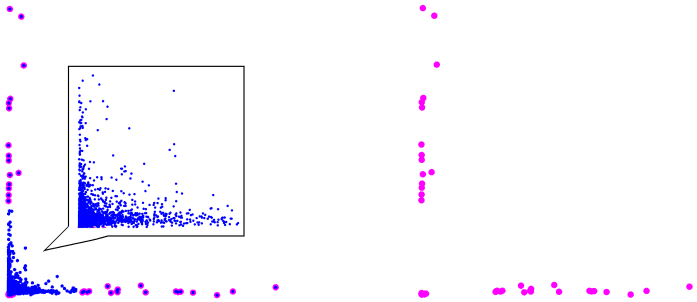
$$\max_{i \in \{1,\ldots,k\}} 1 - \frac{\langle q, \pi_{\leq i}(q, \mathcal{C})\rangle}{\langle q, \pi_{\leq i}(q, \mathcal{O})\rangle}.$$

  This measure is more stringent than what is bounded in (3.1) for the presented definition of approximation; it in fact corresponds to the stronger definition of approximation in (3.2) in Section 3.4.1. To estimate the average error when $\mathcal{Q}$ is large or unknown, $1{,}000$ preferences are randomly chosen and their average is computed.



(a) Annulus ($\alpha = 0.9$), all objects

(b) Annulus ($\alpha = 0.9$), coreset



(c) Stock data, all objects

(d) Stock data, coreset

FIGURE 3.5: Illustration of object workloads.

Experiments were conducted on a Dell OptiPlex 990 with 3.40GHz Intel Core i7-2600 CPU, 8M cache, and 8GB memory.

Workloads. A number of synthetic and real object workloads are used in the experiments. This section chooses to focus on the results for the synthetic ***annulus-uniform*** and ***annulus-clustered***, because they enable testing with a wide range of data characteristics. Objects are drawn from the portion inside the positive orthant of an annulus in $\mathbb{R}^d$ centered at the origin with outer radius 1 and inner radius $\alpha \in [0, 1]$. For annulus-uniform, objects are uniformly distributed inside the annulus. For annulus-clustered, objects are distributed across a mixtures of 20 Gaussians (clipped to the annulus); parameters of the Gaussians allow further control of the clusteredness. For example, Figure 3.5(a) shows a set of objects $\mathcal{O}$ from annulus-uniform with $\alpha = 0.9$, and Figure 3.5(b) illustrates a coreset for $\mathcal{O}$.

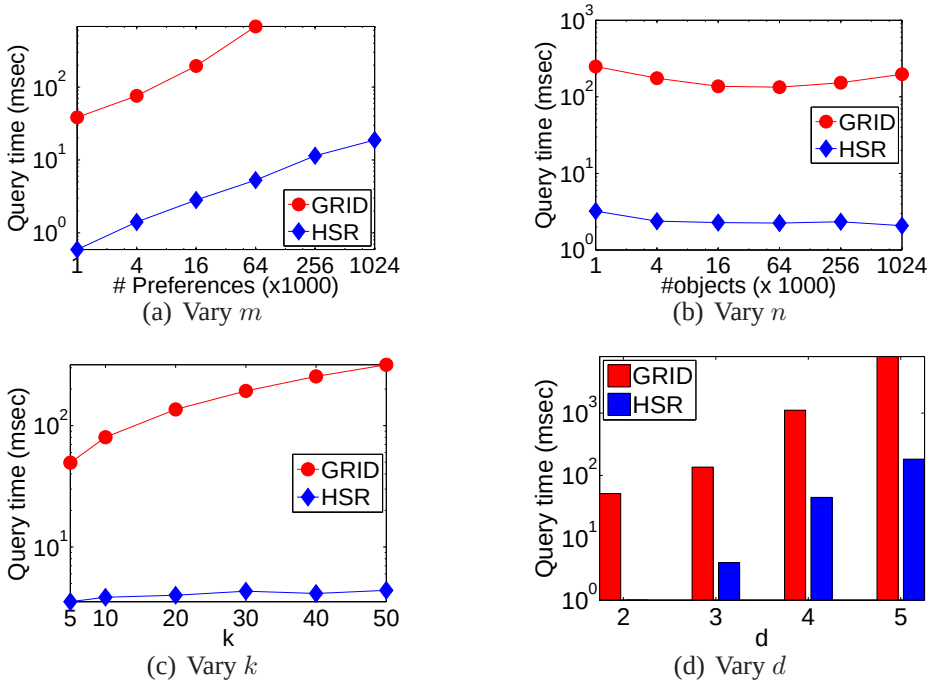
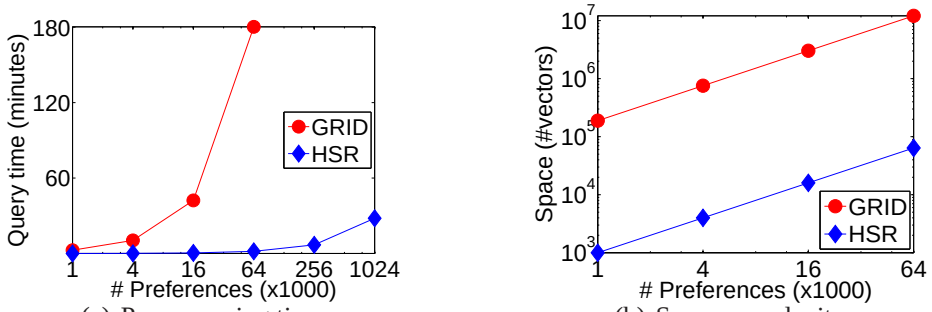


FIGURE 3.6: Comparison between HSR and GRID (previous approach) for static reverse top-k queries; $\alpha = 0.9$.

To generate an object update for the workload, either insertion or deletion is first chosen with equal probability. For insertion, a new object is drawn from the same distribution used to draw the initial object set. For deletion, an existing object is chosen at random with equal probability.

Annulus-uniform and annulus-clustered are related to the synthetic workloads (*correlated*, *anti-correlated*, and *uniform*) described in [28]. Note that α gives us some control over the "hardness" of the problem. As α approaches to 0, more and more objects, particularly those closer to the origin, do not participate in any top-k lists. The object distribution becomes uniform inside the ball. It remains harder for the problem than uniform distribution inside the unit box, for which only few objects close to the corners of the unit box participate in any top-k lists. The distribution of objects for annulus-clustered, generated with one single Gaussian, resembles correlated. As α approaches 1, the objects lie on the sphere, the distribution of objects becomes more anti-correlated, and any object can appear



(a) Preprocessing time (b) Space complexity

FIGURE 3.7: Additional scalability comparison between HSR and GRID.

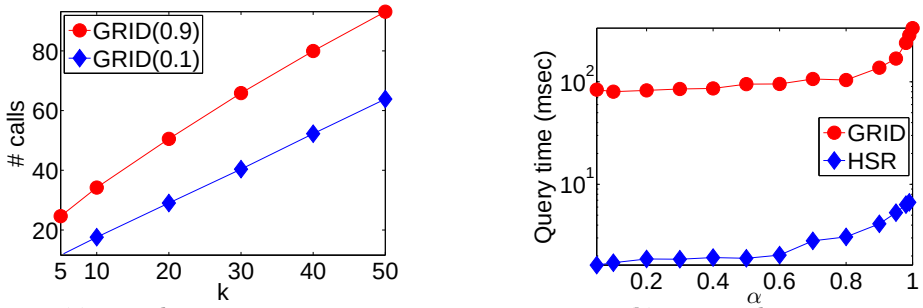
in a top-k list; this is in some sense captures the worst-case behavior.

In addition to synthetic object workloads, data for $2{,}374$ stocks on NYSE and NASDAQ from Yahoo! Finance were obtained. For each stock, its estimated earnings per stock (EPS) and weekly historical quotes (opening, closing, lowest, and highest prices and volume) in 2011 were collected. EPS can be used to convert each price to a price-to-earning ratio (PER), which is a more normalized metric than raw price for comparing different stocks. In the experiments only 2 dimensions are used: volume, and PER based on the closing price (although PER can be generated from various available prices, they would be extremely similar). Figure 3.5(c) shows the objects from this dataset, and Figure 3.5(d) shows its coreset. Note that the extreme points are well-represented in the coreset, while the cluster near the origin requires few representatives.

Preferences are generated from one of the following two distributions. With ***Uniform***, preference is drawn uniformly at random from the unit sphere $\mathbb{S}^{d-1}$ inside the positive orthant of $\mathbb{R}^d$. With ***Clustered***, preferences are distributed across a mixtures of 20 Gaussians over the sphere; parameters of the Gaussians allow further control of the clusteredness.

3.5.1 Static Reverse Top-k Queries

First, we compare HSR with GRID [115]. Unless specified otherwise, $d = 3$, $m = 10{,}000$, $n = 10{,}000$, and $k = 20$ in this section. The objects are generated from annulus-uniform with $\alpha = 0.9$ (unless specified otherwise). One thousand query objects are drawn from the



(a) Vary k; $\alpha = 0.9$ or 0.1 (b) Vary α; $k = 20$

FIGURE 3.8: More comparison between HSR and GRID.

same distribution.

Figures 3.6(a) and 3.6(b) compare the average query time as m (the number of preferences) and n (the number of objects) increase, respectively. HSR, which uses a linear-size data structure, performs one to two orders of magnitude better than GRID. Results are not shown for GRID when $m = 256{,}000$ and $1{,}024{,}000$, because it becomes too expensive. HSR's scalability advantage over GRID is reflected not only in terms of query time, but also preprocessing time (Figure 3.7(a)) and space consumption (Figure 3.7(b)). Preprocessing for GRID involves reordering of preferences and many reverse top-k computations for materialized views, and it takes more than 3 hours for $m > 64{,}000$. GRID also consistently uses two orders of magnitude more space than HSR; here, the space of GRID is measured by the number of preferences it materializes, and the space of HSR by the number of cutoff points it indexes, both of which are d-dimensional vectors.

Figure 3.6(c) shows that when k increases, the average query time increases for GRID but remains almost the same for HSR. The reason is that the number of top-k queries made by GRID depends on k, which becomes clear when we examine Figure 3.8(a). Figure 3.8(a) shows the number of top-k queries made by GRID for $\alpha = 0.9$ (the workload used by Figure 3.6(c)), and the number for $\alpha = 0.1$. Both exhibit growth linear in k, and we see that $\alpha = 0.9$ is indeed "harder" than $\alpha = 0.1$. On the other hand, HSR always issues one halfspace range query per request (and therefore it is not shown in Figure 3.8(a)),

regardless of k and α.

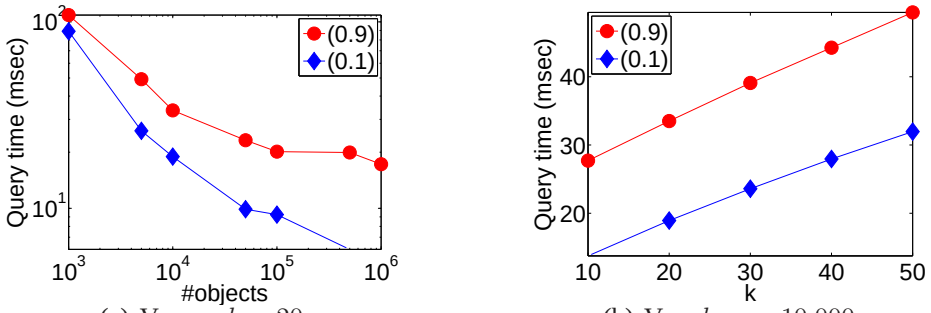
Figure 3.6(d) shows how HSR and GRID perform as the dimensionality increases. We see that the advantage of HSR over GRID is maintained as d increases. For $d = 2$, HSR answers a reverse top-k query in 0.38 milliseconds on average, which, if shown in Figure 3.6(d), would have been below the horizontal axis (note the log-scaled vertical axis).

Figure 3.8(b) summarizes the comparison between HSR and GRID when varying α, the inner radius of the annulus. The query time generally increases as the annulus becomes thinner (approaching a sphere), but HSR maintains its lead over GRID across all α values.

Figures 3.9 shows the performance of HSR when the number of preferences scales up to one million. GRID becomes too slow to run in this case. Two curves are shown in Figure 3.9(a): one for $\alpha = 0.1$ and one for $\alpha = 0.9$. For $\alpha = 0.1$, the algorithm performs better when n is bigger, because more objects actually lower the chance that a query object becomes relevant to the preferences. Figure 3.9(b) shows the average query time slightly increases as k increases. Compared with Figure 3.6(c), the average query time increases by a factor of 10 when the number of preferences increases by a factor of 100. The reason is that the size of the index for halfspace range queries depends on the number of preferences.

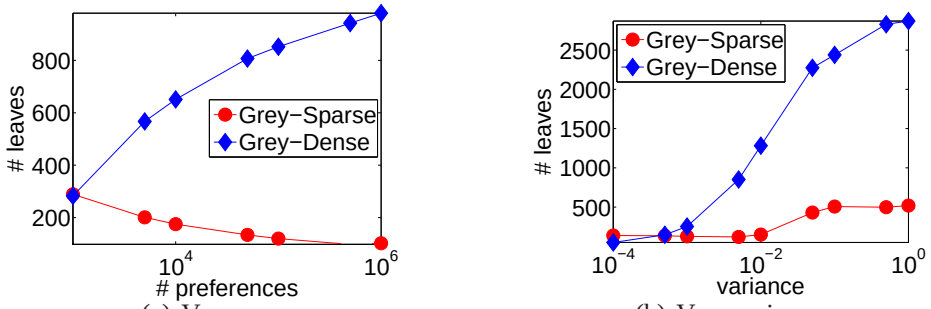
3.5.2 Continuous Top-k Queries

An object update is costly for GRID because a lot of materialized views need to be recomputed for the object update. Many top-k queries are called as subroutines even if the object update does not affect any preference's top-k results. For annulus-uniform with $\alpha = 0.8$, $d = 2$, $m = 10{,}000$, $n = 1{,}000$, and $k = 10$, the average update time of GRID is 40 seconds. In comparison, HSR takes only 0.014 seconds per update. Since the performance of HSR clearly dominates that of GRID for continuous top-k queries, GRID is omitted in the remainder of this section. Unless specified, $\alpha = 0.8$, $k = 10$, $d = 2$, objects are



(a) Vary n; $k = 20$ (b) Vary k; $n = 10{,}000$

FIGURE 3.9: Reverse top-k query on 1 million preferences; $d = 3$.



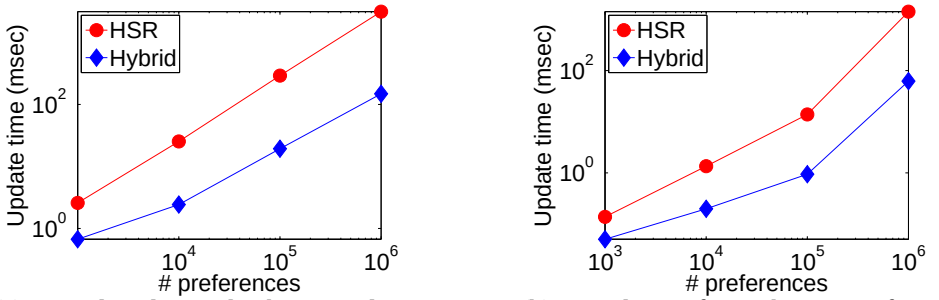
(a) Vary m (b) Vary variance

FIGURE 3.10: Numbers of grey-sparse and grey-dense leaves in $\mathcal{T}$.

drawn from annulus-uniform, and preferences generated from the clustered distribution. For hybrid approach, both τ_m and τ_n are set to 1 (recall Section 3.3.2.3).

We first see how the hybrid approach automatically adapts to the object and preference workloads. Figure 3.10(a) shows that as the number of preferences increases, more grey-sparse nodes in $\mathcal{T}$ are converted into grey-dense nodes. Those preferences in grey-dense nodes are not processed individually, making hybrid approach more scalable to a large number of preferences. Figure 3.10(b) shows the number of grey-sparse and grey-dense leaves when varying the variance of the Gaussian distributions from which preferences are generated. When variance is small, many parts of the QRS have few or no preferences, so hybrid uses fewer grey-dense nodes and takes a preference-driven approach for these parts.

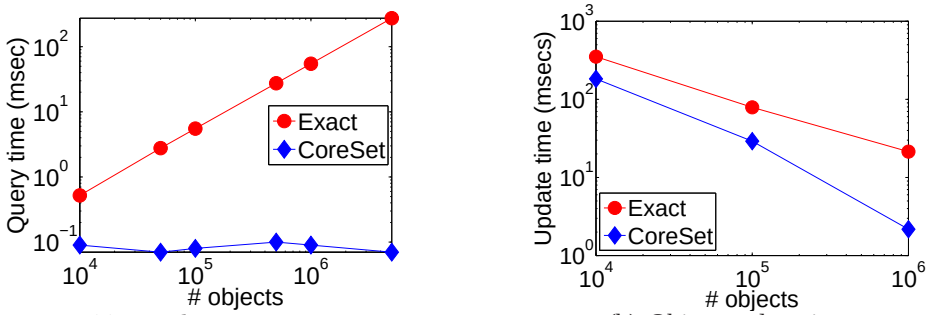
Next, we compare the preference-driven and hybrid approaches for continuous top-



(a) Annulus-clustered objects, clustered preferences

(b) Annulus-uniform objects, uniform preferences

FIGURE 3.11: Preference-driven vs. hybrid.

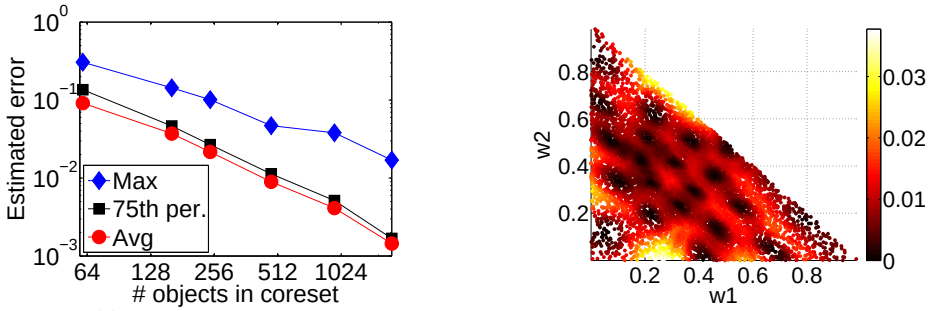


(a) Top-k query time

(b) Object update time

FIGURE 3.12: Exact vs. coreset-based approximation.

k queries. The number of objects is set to $1{,}000$, and the number of preferences varies from $1{,}000$ to one million. Figures 3.11(a) and 3.11(b) compare the performance of the preference-driven (denoted HSR in figures) and hybrid approaches for annulus-clustered and annulus-uniform, respectively; preferences are drawn from clustered and uniform, respectively. The combinations of (annulus-clustered objects, uniform preferences) and (annulus-uniform objects, clustered preferences) are omitted because they show similar trends. For these workloads, when the ratio between the number of preferences and the number of objects becomes large, the hybrid approach performs significantly better than the preference-driven one, as it avoids multiple computations for many preferences sharing the same k-th ranked object.



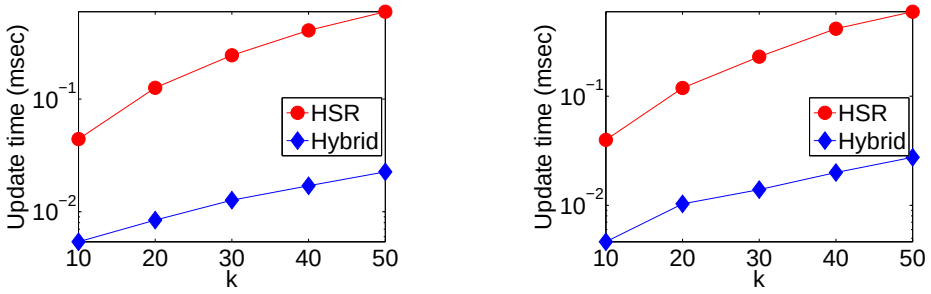
(a) Vary coreset size (b) Error distribution; $|\mathcal{C}| = 509$

FIGURE 3.13: Approximation error; $|\mathcal{O}| = 100{,}000$.

3.5.3 Continuous Approximate Top-k Queries

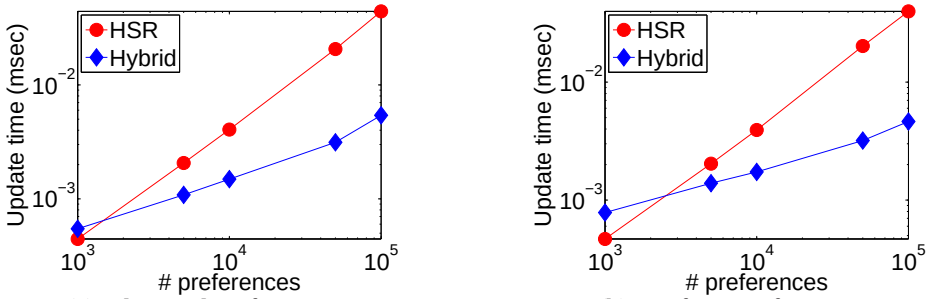
In this section, the number of preferences is set to $100{,}000$; $k = 20$, $d = 3$, and $\alpha = 0.9$. Figure 3.12 shows the effect of the number of input objects on the coreset-based approximation algorithm, in comparison with the exact one; here, the size of the coreset is fixed roughly at $1{,}000$. Figure 3.12(a) shows the top-k query time when the number of objects varies from $10{,}000$ to one million. While the query time for the exact algorithm increases, it remains roughly the same for the coreset-based algorithm, because the size of the top-k index is proportional to $|\mathcal{C}|$ instead of $|\mathcal{O}|$. Since an insertion or deletion of a preference involves a top-k query, the coreset-based algorithm will be able to handle preference updates better than the exact one for a large set of objects. Figure 3.12(b) shows the processing time per object update when the number of objects varies from $10{,}000$ to one million. The gap between the performances of the exact and coreset-based approximation algorithms widens as the number of objects increases, because 1) when $|\mathcal{O}|$ becomes large, most object updates would not affect the coreset if an object update is randomly chosen, and 2) the top-k query can be answered more efficiently on a smaller coreset.

Figure 3.13(a) shows how the size of the coreset affects the quality of the approximation. As expected, the larger the coreset, the higher the accuracy. By choosing roughly 500 objects in the coreset, the estimated maximum and average errors are less than 0.05 and 0.01, respectively. Moreover, majority of the errors are small, as indicated by the closeness



(a) Clustered preferences (b) Uniform preferences

FIGURE 3.14: Preference-driven vs. hybrid: Yahoo! Finance data; $m = 100{,}000$.



(a) Clustered preferences (b) Uniform preferences

FIGURE 3.15: Preference-driven vs. hybrid: Yahoo! Finance data; $k = 10$.

between the average error and error at the 75th percentile. Figure 3.13(b) further plots the distribution of errors over the preferences in $\mathbb{Q}$. Preferences at the boundary tend to have slightly higher approximation errors.

3.5.4 Yahoo! Finance Data

Experiments in this section study the performance of the exact algorithms (preference-based and hybrid) on the object (stock) data collected from Yahoo! Finance. Since user data are not disclosed to the public, synthetic preferences generated from clustered and uniform are used in the experiments. While the distribution of objects and update workload are considerably different from the synthetic ones (as illustrated in part by Figure 3.5), performance results are similar to those in Section 3.5.2.

Figure 3.14 shows the average update time as k increases. As expected, the number of affected preferences increases as k increases. For the preference-based approach, a top-k

query is called for each affected preference. For the hybrid approach, the complexity of the k-level depends on the value of k, and a larger k increases the size of the search tree $\mathcal{T}$. Figure 3.15 shows the average update time as the number of preferences increases. Similar to the synthetic workloads (Figure 3.11), all curves exhibit growth proportional to m, because the number of affected preferences increases as m increases. For a small number (up to around a thousand) of preferences, the preference-based approach may be more attractive because of the overhead of hybrid's flexibility and the small workload size in this case, but hybrid remains the better choice if m is reasonably large.

3.6 Related Work

There is a large body of literature on top-k query processing (see [69] for a survey); much of it concerns the linear preference top-k queries and variants [44, 67, 113, 85, 50, 49, 115] that are considered in this chapter. This section elaborates on three pieces of work that are most related to the ones presented in this chapter.

Das et al. [49] considered the problem of supporting ad hoc (i.e., non-continuous) top-k queries over streams. They also took a geometric approach and developed a data structure based on maintaining an arrangement of lines in the dual space. Their solution uses halfspace range queries as a primitive, but not for the purpose of solving reverse top-k queries as proposed in this chapter. Time and space complexities are improved by pruning the set of objects to a superset of the k-skyband, which is computed by partitioning the arrangement into "strips" and using the top-k query results for the borders of the strips to prune dual lines from each strip. The query and update operations take linear time, and heuristics are required in choosing the partitioning. There was some discussion on the case of $d > 2$, but the solution was only evaluated for $d = 2$. In comparison, the coreset-based approach to approximating the k-level provides guarantees and generalizes to higher dimensions.

Mouratidis et al. [85] proposed the *TMA* algorithm (and the more specialized *SMA*) for supporting multiple continuous top-k queries over data streams. TMA partitions the primal space into grids, and for each cell, stores an "influence list" of queries (those returned by a reverse top-k query with the cell's top-right corner). Given an object update, TMA identifies affected queries by searching for affected cells in an order that minimizes the number of cells visited. Since TMA materializes the top-k answer for each query, it requires more space than the approach of recording only the cutoff points. They also target fewer number (thousands) of queries than the work presented in this chapter (hundreds of thousands). A direct comparison with the work presented in this chapter is difficult because TMA and SMA also have features specific to the object update pattern under the sliding-window semantics.

Most relevant to this chapter is the work by Vlachou et al. [115]. Their *monochromatic* reverse top-k algorithm can compute, without knowing the actual preferences, a description of the set of possible preferences that would be affected by a given object. The algorithm works for $d = 2$, based on similar observations as *ranked join indices* [113]. The QRS-based framework and techniques can solve the same problem in higher dimensions as well; see Section 3.7 for more details. For the *bichromatic* reverse top-k problem, where the set of preferences is given, two algorithms were proposed. *RTA* heuristically orders the preferences to be processed based on similarity, to increase the chance that the top-k query result for the current preference can be reused for the next preference. *RTOP-Grid* uses a grid data structure for pruning. For each cell, a reverse top-k query is run for the lower-left and upper-right corners, and the result lists are stored in the cell. These lists are used to reduce the set of preferences to be further evaluated using RTA. RTOP-Grid provides no theoretical performance guarantees, and object updates are particular expensive for the grid data structure. The experimental evaluation in Section 3.5 compares RTOP-Grid with the solutions presented in this chapter.

3.7 Conclusion and Other Applications

This chapter studied the problem of scalably processing a large number of continuous top-k queries, each with a different preference vector for ranking multi-attribute objects. The notion of QRS (query response surface) was proposed and the solutions were developed within a geometric framework. By recognizing the connection to halfspace range queries, data structures were obtained for reverse top-k queries with linear space and sublinear query time. Building on this result, a fully dynamic solution was developed to support both object and preference updates efficiently. This chapter also defined and solved an approximate version of the problem, further improving efficiency with little loss of accuracy. Experimental evaluation confirmed the effectiveness of the presented ideas such as selective QRS-driven processing and coreset-based QRS simplification, which helped advance the presented solutions in both scalability and functionality.

In closing, we briefly discuss several settings beyond those focused on by this chapter, where the techniques presented in this chapter may be applicable.

Reverse k-nearest-neighbor queries have been widely studied by the database community; see [90] for an overview. Though these queries are not the focus of this chapter, they can also be handled by the approach presented in this chapter. More precisely, a set $\mathcal{O}$ of points in $\mathbb{R}^d$ can be mapped to a set $\hat{\mathcal{O}}$ of points in $\mathbb{R}^{d+1}$ so that the k-nearest-neighbor query for a point $q \in \mathbb{R}^d$ can be formulated as the top-k query for a preference $\hat{q} \in \mathbb{S}^d$; more details will be provided in the generalization section in Chapter 5.5.

In some settings, the set $\mathcal{Q}$ of preferences is not given explicitly. Instead, given an object update, we are interested in obtaining (a description of) the set of all possible preferences affected by it. This query is termed *monochromatic reverse top-k* by [115], with applications in business analysis [115] and in publish/subscribe systems using the *message reformulation paradigm* [41]. The concept of QRS and the QRS-driven approach in Section 3.3.2 offer a solution that generalizes to high dimensions. Maintaining the full QRS

is expensive, however. When approximation is acceptable, the coreset-based approach in Section 3.4 can simplify the QRS, improve running time, and reduce the complexity in describing the affected preferences.

Recently, there is growing interest in handling uncertainty in preference vectors and assessing sensitivity in ranking to perturbations in preferences [108]. The notion of QRS provides a natural framework for these problems, and the coreset-based approximation can be readily applied to improve solution scalability. Further investigation would be a promising direction of future work.

Finally, preference top-k queries also have applications in information retrieval (where, e.g., a multi-keyword search can be seen as a preference top-k query over documents in a high-dimensional keyword vector space) and in information integration (where results from multiple sources are merged and ranked according to a preference function). Recent work [70] has studied how to share the work involved in processing multiple such queries. It would be interesting to investigate whether the techniques presented in this chapter can be applied help improve scalability in a complementary manner.

4

Top-k Preferences in High Dimensions

This chapter extends the solution in the previous chapter, which is effective only in low dimensions, to much high dimensions (in up to high tens). The solution presented in this chapter is efficient if many preferences exhibit *sparsity*—i.e., each specifies non-zero weights for only a handful (say 5–7) of attributes (though the subsets of such attributes and their weights can vary greatly). The main idea is to carefully select a set of low-dimensional *core subspaces* to "cover" the sparse preferences in a workload. These sparse preferences can be indexed more effectively in these subspaces than in the full-dimensional space. Being multi-dimensional, each subspace covers many possible preferences; furthermore, multiple subspaces can jointly cover a preference, thereby expanding the coverage beyond the dimensionality of each subspace. Experimental evaluation validates the effectiveness of the solution presented in this chapter and its advantages over previous solutions.

4.1 Introduction

Challenge: curse of dimensionality. Supporting linear preference top-k queries and the reverse top-k queries becomes challenging for high dimensions (say 40). For preference

top-k queries, the Threshold Algorithm (TA) [58] is efficient if every top-k object is ranked high in at least one dimension. However, as the dimensionality d grows, there is a higher chance that an object has a low rank even if it ranks high along one dimension. The layer-based approach, represented by [44], indexes layers of convex hulls for the objects in the full-dimensional space; computing a convex hull takes $O(n^{\lfloor d/2 \rfloor} + n \log(n))$ time, and the outer layers grow in size quickly with d. The view-based approach [68, 50] uses a set of materialized top-k views to compute top-k queries, but in high dimensions, a large number of materialized views are required to provide adequate support for queries. Recently, Heo et al. [65] combined the layer-based technique with TA-style dimension-wise filtering for top-k queries involving arbitrary subset of attributes. All work mentioned above tested no more than 7 dimensions.

For reverse top-k queries, the approach of [115] reduces a reverse top-k query to m top-k queries, where m is the number of preferences in the worst case. Chapter 3 uses a duality approach to construct a linear-size index that can answer a reverse top-k query in sublinear time given fixed dimensionality d. For low dimensions ($d \leq 3$), the query time is $O(\log m + k)$, which is optimal. Although the solution is scalable in the number of preferences, support for reverse top-k queries in high dimensions is still inadequate. The duality approach reduces a reverse top-k query to halfspace reporting, whose tradeoff between query time and space complexity has been studied [83]. If the storage requirement is near-linear, say $O(n\text{polylog}(n))$, then the query time of best known algorithms is $\Omega(n^{1-1/\lfloor d/2 \rfloor} + t)$ [83], where t is the number of results, and the hidden constant of proportionality is exponential in d. Furthermore, these algorithms are too complex to implement. For practical data structures such as quad-trees and kd-trees, a halfspace query requires $\Omega(n)$ time in the worst case and roughly $O(n^{1-1/d} + t)$ for uniformly distributed points. Hence, for high-dimensional data, existing approaches will not outperform a simple linear scan.

Opportunity: sparse preferences. In practice, even if data have high dimensionality, users are usually interested in only a small subset of attributes—few users are able to specify preferences with non-zero weights for a large number of attributes in a meaningful way. Thus, there is an opportunity to develop techniques for handling such "sparse" preferences differently from and more efficiently than the general case. If many preferences are sparse, overall performance can be greatly improved by speeding up the common case.

This observation and the techniques to be presented in this chapter differ from the existing dimensionality reduction techniques, such as principal component analysis (PCA), random projection, and low-distortion embedding techniques, which are usually applied to the object set. It is arguable that reducing object dimensionality alone is neither a perfect or a complete solution: While these methods are effectively in projecting data to moderate dimensions, say 100's to 10's, using these methods to project objects onto 5–7 dimensions can create significant error. Therefore, objects need to be projected on multiple subspaces if we wish to work with low-dimensional spaces. Also, attributes in the reduced space are harder for users to work with as they may no longer have intuitive meanings. Although preferences in the original space can be mapped to ones in the reduced space, they may become more difficult to handle because they may no longer retain their sparsity. The idea of dimensionality reduction is, in fact, used in this chapter, but objects and preferences are jointly considered in a careful way to avoid these above problems. Moreover, our techniques are still applicable even if the objects cannot be embedded into low-dimensional spaces.

Approach and contributions. This chapter presents efficient data structures and algorithms for top-k and reverse top-k queries in high dimensions. Our approach is effective when most of the preferences are sparse—i.e., each of them specifies non-zero weights for only a small number (say 2–6) of attributes (but they do not need to specify the same subset of attributes or similar weights on attributes). For top-k queries, in order to take ad-

vantage of sparsity in query preferences, our approach needs to assume the distribution of *which* attributes are specified by the preferences, but the approach still works well without accurate knowledge of the distribution of *what weights* are specified by the preferences for these attributes.

Roughly speaking, this chapter follows a dimension-reduction framework, but objects and preferences are not projected on a single low-dimensional subspace. Instead, they are projected on many subspaces. For each subspace, an index is built on a subset of objects. To answer a top-k or reverse top-k query, only a small number of subspaces is chosen; a low-dimensional query is performed on each of them and then their results are combined to answer the overall query. In addition, approximation methods are used to reduce the size of the index and to expedite the query procedure. Experimental evaluation confirms the effectiveness of our approach, which allows a desktop machine to handle hundreds of thousands of objects or preferences in 20 to 100 dimensions with speed and accuracy. To the best of our knowledge, our approach is the first to demonstrate this degree of scalability in both problem size and dimensionality.

Outline of solution. In more detail, a set $\mathbb{H}$ of low-dimensional subspaces, called *core subspaces*, is carefully chosen based on the given distribution of preferences. For each core subspace $H \in \mathbb{H}$, a small subset of objects that are "relevant" for H is chosen and projected on H. Let $\mathcal{O}_H$ denote the resulting projections. Building on the techniques for handling low-dimensional preferences in Chapter 3, $\mathcal{O}_H$ is indexed for each H. To answer a top-k query with respect to a sparse preference q, a small subset $\Gamma_q \subset \mathbb{H}$ of core subspaces, which "cover" the query preference q, is chosen. For each $H \in \Gamma_q$, the top-βk ranked objects of $\mathcal{O}_H$ are computed for a parameter $\beta \geq 1$, with respect to the preference q (or rather w.r.t. the projection of q on H). Finally, the top k objects are returned among the union of these objects.

To support reverse top-k queries for a set $\mathcal{O}$ of objects and a set $\mathcal{S}$ of preferences, each

preference $q \in \mathcal{S}$ is assigned to a small subset of $\Gamma_q \subset \mathbb{H}$ of core subspaces that cover q. For each core subspace $H \in \mathbb{H}$, let $\mathcal{S}_H$ denote the projections on H of preferences assigned to H. $\mathcal{S}_H$ is indexed to support reverse top-βk queries against $\mathcal{O}_H$ and $\mathcal{S}_H$. To answer a reverse top-k query for a query object o, the core subspaces that are "relevant" for o are identified; a reverse top-βk query with o is performed on each of them; all result preferences are collected, and false positives are filtered out.

Technical challenges. There are several technical challenges that need to be addressed to complete this solution. First, how are the core subspaces chosen? A naive approach will be to make any subspace that contains some preferences to be a core subspace. For example, if preferences specify non-zero weights for attribute subsets $\{1, 2\}$, $\{1, 3\}$, and $\{2, 3, 4\}$, then they are selected as core subspaces and indexes are built for them: 2-dim indexes for $\{1, 2\}$ and $\{1, 3\}$, and 3-dim for $\{2, 3, 4\}$. This approach is not practical, however, because there are too many possible low-dimensional subspaces. For example, if objects have 20 attributes and each preference specifies at most three of them, one might have to build $\binom{20}{3} = 1{,}140$ different indexes.

Another possibility is to cluster the preferences into a small number of clusters and choose a representative preference, called a *view*, from each cluster. This view-based approach [68, 50] works if preferences are tightly clustered, objects are "well distributed," and the weights of query preferences for top-k queries follow the same distribution of $\mathcal{S}$. As we will see later, this approach does not always work well, because each view is very "specific" and many more views will be needed as dimensionality grows. This chapter shows how to overcome the limitations of this approach with multi-dimensional core subspaces, each of which effectively serves as a "super"-view that subsumes an infinite number of preference-based views lying in it. Section 4.3 describes this core-subspaces approach.

Second, it will be too expensive to build an index on the entire set of objects for each

core subspace, so Section 4.4.1 describes a method, which builds on the results in Chapter 3, for choosing a small set of objects to index. Analogously, it is expensive to index all preferences in each core subspace, so Section 4.4.2 introduces a method for assigning each preference to a small number of core subspaces where it will be indexed. Then, using the indexes described in Section 4.4, Section 4.5 shows how to answer top-k and reverse top-k queries.

Finally, we cannot assume that all preferences are sparse or all can be covered by the selected core subspaces. Therefore, Section 4.4.3 shows how to build full-dimensional indexes for uncovered preferences. In particular, Section 4.4.3 describes an approximation method similar to the one in [7], but with an improvement: if input objects lie on a low-dimensional surface, say of dimension τ, then one can choose a subset $\mathcal{C}$ of objects whose size is exponential only on τ, but polynomial in d, which provides top-k query answers that approximate those obtained by querying the entire set of objects.

4.2 Preliminaries

Note that the algorithms for answering preference and reverse top-k queries [1] in a low-dimensional space have been presented in Chapter 3; this chapter will use them as black-boxes to solve the high-dimensional case. Note that Chapter 3 also formally defines the coreset for answering approximate preference top-k queries. In addition to the geometric concepts (duality transform, arrangement, coreset, and top-k query response surface) introduced in Chapters 2 and 3.2.2, the following concepts will be used throughout this chapter:

Span. Let the x_i-axis represents the i-th attribute. Let e_i denote the unit vector in direction x_i, i.e., the i-th coordinate of e_i is 1 and the rest are 0. A subset $I \subseteq [1, d]$ of attributes defines an axis-parallel subspace $\mathsf{Sp}(I)$ of $\mathbb{R}^d$ in which only the attributes of I have

[1] Both preference top-k query and reverse preference top-k query are formally defined in Chapter 3.

non-zero values. Formally, $\mathsf{Sp}(I) = \{\sum_{j \in I} \lambda_j e_j \mid \lambda_j \in \mathbb{R}\}$. For two axis-parallel subspaces $H_1 = \mathsf{Sp}(I_1)$ and $H_2 = \mathsf{Sp}(I_2)$, let $\mathrm{span}(H_1, H_2)$ denote the smallest axis-parallel subspace that contains both H_1 and H_2; equivalently, $\mathrm{span}(H_1, H_2) = \mathsf{Sp}(I_1 \cup I_2) = \{\lambda_1 x_1 + \lambda_2 x_2 \mid x_1 \in H_1, x_2 \in H_2, \text{ and } \lambda_1, \lambda_2 \in \mathbb{R}\}$.

Sparse preference. Recall that a *preference* is represented as a unit vector in $\mathbb{R}^d$. , i.e., a point $(w_1, \ldots, w_d)$ on $\mathbb{S}^{d-1}$, the $(d-1)$-dimensional unit sphere embedded in $\mathbb{R}^d$. In this chapter, each $w_i \in [-1, 1]$ is the *weight* for the i-th attribute (weights can be negative). For a preference q, $\mathsf{Sp}(q)$ is defined to be the subspace spanned by the non-zero attributes of q. Note that $\dim(\mathsf{Sp}(q))$ may be much smaller than d. For example, if $q = (1/\sqrt{2}, 1/\sqrt{2}, 0, \ldots, 0)$, then $\mathsf{Sp}(q)$ is the 2-dimensional x_1x_2-plane.

Projection on subspace. q is $\langle q, o \rangle = \sum_{1 \le i \le d} w_i v_i$. For a point $x \in \mathbb{R}^d$ and an axis-parallel subspace H, let x_H denote the projection of x on H. For example, if $x = (x_1, \ldots, x_d)$ and H is spanned by attributes $\{1, 2, 4\}$, then $x_H = (x_1, x_2, x_4)$. Recall that the *score* of an object o with respect to a preference q is $\langle q, o \rangle = \sum_{1 \le i \le d} w_i v_i$. For a preference q and an object o, $\langle q, o \rangle = \langle q_H, o_H \rangle$ where $H = \mathsf{Sp}(q)$; in other words, when computing the score of o w.r.t. q, it suffices to do so for their projections on the subspace $\mathsf{Sp}(q)$.

4.3 Identifying Core Subspaces

This section describes the algorithm for computing the set $\mathbb{H}$ of core subspaces, which is used to build low-dimensional indexes. For these indexes to be practically efficient, the maximum dimensionality of a core subspace is capped at $\hat{\tau} = 5$.

Let $\mathcal{S}$ be a set of preferences. It can be a set of given preferences for reverse top-k queries, or a past workload of forward top-k queries that can be used to inform index construction.

The algorithm works in three stages. The first stage identifies the initial set $\mathbb{K}$ of *candidate* subspaces from the "sparse" preferences of $\mathcal{S}$ (the formal definition of "sparseness" will follow shortly). If $\mathbb{K}$ is small, let $\mathbb{H} = \mathbb{K}$ and the algorithm terminates. Otherwise, the algorithm proceeds to the next stage, adding to $\mathbb{K}$ a few additional subspaces that span multiple subspaces of $\mathbb{K}$ and are "popular" (roughly speaking, a popular subspace can help "cover" many sparse preferences—the notion of "coverage" is intuitive but will be made more clear in Section 4.4.2). The last stage chooses a subset of $\mathbb{K}$ to cover most of the sparse preferences of $\mathcal{S}$. The remainder of this section is devoted to describe each stage in detail.

Initializing candidate subspaces. For the purpose of finding core subspaces, the algorithm ignores insignificant attribute weights in preferences. Consider each preference $q \in \mathcal{S}$. The algorithm rounds off any attribute weight to 0 if no greater than 0.01 (which would decrease $\|q\|$, the L_1-norm of q, by no more than 1%), and rescales the resulting preference so that it remains a unit vector.

Following this preprocessing, a preference q is said to be τ-*dense* if $\dim(\mathsf{Sp}(q)) \leq \tau$ (i.e., q has non-zero weights for at most τ attributes). Since the algorithm is practically limited to core subspaces with dimensionality up to $\hat{\tau} = 5$, it focuses on the subset $\mathcal{S}_s$ of *sparse preferences*, i.e., those that are $(\hat{\tau} + \triangle\tau)$-dense. Here, $\triangle\tau$ is a small slack ($\triangle\tau$ is set to 2) that reflects the ability of the core-subspace approach to handle denser preferences using multiple core subspaces.

The set $\mathbb{K}$ of candidate core subspaces is computed from the set $\mathcal{S}_s$ of sparse preferences as follows. First, any $\hat{\tau}$-dense preference gives us an axis-parallel candidate subspace: $\mathbb{K} \leftarrow \{\mathsf{Sp}(q) \mid q \in \mathcal{S}_s \text{ and } q \text{ is } \hat{\tau}\text{-dense}\}$. Second, for each sparse preference $q \in \mathcal{S}_s$ that is not $\hat{\tau}$-dense (but still $(\hat{\tau} + \triangle\tau)$-dense), all $\hat{\tau}$-dimensional axis-parallel subspaces of $\mathsf{Sp}(q)$ are considered as candidates: $\mathbb{K} \leftarrow \mathbb{K} \cup \{\mathsf{Sp}(I) \mid \mathsf{Sp}(I) \subset \mathsf{Sp}(q) \text{ and } |I| = \hat{\tau}\}$.

If the size of $\mathbb{K}$ is small, the algorithm sets $\mathbb{H}$ to $\mathbb{K}$ and stops, otherwise, it proceeds to

the next two stages. As mentioned in Section 4.1, however, $\mathbb{K}$ can be large. For example, for $d = 20$, $\hat{\tau} = 5$, and $\triangle\tau = 2$, $|\mathbb{K}|$ can be as large as $21{,}699$.

Adding popular subspaces. To capture the notion of "popularity," the *weight* of a subspace H (with respect to the set of sparse preferences $\mathcal{S}_s$) is defined as

$$w(H) = \sum_{q \in \mathcal{S}_s} \|q_H\|^2 / (\dim(H))^\mu, \tag{4.1}$$

where q_H denotes the projection of q on H, and μ is a parameter (further explained below). Intuitively, the weight function favors those subspaces that have low dimensionality but preserve most information about preferences, in the sense that $\|q_H\|$ is large.

$\|q_H\|^2$ is chosen instead of $\|q_H\|$ in this definition, because we wish to reward subspaces that preserve most information about a preference (i.e., $\|q_H\|$ is close to 1), and penalize those that preserve little information about a preference (i.e., $\|q_H\|$ is close to 0). For example, given two preferences, consider 1) two subspaces, where each contains one preference (whose projection has norm of 1) but is orthogonal to the other preference (whose projection has norm 0), versus 2) two subspaces for which both preferences have projections of norm 0.5. Intuitively, the two subspaces in the first case are better because they provide "full coverage" for each of the two preferences, while the two subspaces in the second case only provide "partial coverage" for both preferences. The presented weight definition captures this intuition with the use of $\|q_H\|^2$. Had $\|q_H\|$ been used instead, these subscriptions would have identical weights.

If all preferences in $\mathcal{S}_s$ lie within H, then $w(H) = |\mathcal{S}_s|/(\dim(H))^\mu$, which is the maximum possible weight for subspaces with the same dimensionality. The term $(\dim(H))^\mu$ penalizes high-dimensional subspaces because constructing indexes for them is more expensive than for low-dimensional subspaces. The term also serves to "normalize" popularity, because a high-dimensional subspace is expected to be able to cover more preferences.

By adjusting the parameter μ, a trade-off is obtained between keeping the indexing costs low and covering more preferences. μ is set to $\frac{1}{4}$ for experiments in Section 4.6.

We are now ready to describe how to add popular subspaces to $\mathbb{K}$. Suppose $\mathbb{K}$ has two overlapping subspaces of significant weights. It might be more efficient to build a single index for $\mathrm{span}(H_1, H_2)$ rather than building two separate indexes—one for H_1 and another for H_2. To enable this possibility, given $H_1, H_2 \in \mathbb{K}$, $H = \mathrm{span}(H_1, H_2)$ is added to $\mathbb{K}$ if all following conditions hold:

- $\dim(H) < \dim(H_1) + \dim(H_2)$; i.e., H_1 and H_2 overlap.
- $w(H_1)$, $w(H_2) \geq \mathrm{median}\{w(K) \mid K \in \mathbb{K}\}$, and $w(H) \geq 0.8(w(H_1) + w(H_2))$; i.e., the subspaces considered are sufficiently popular.
- $\dim H \leq \hat{\tau}$, where $\hat{\tau}$ is maximum dimensionality of a core subspace (introduced at the beginning of this section); the algorithm does not consider adding subspaces with higher dimensionality, because indexing them would be too costly.

The addition of popular subspaces is implemented by sorting $\mathbb{K}$ in decreasing order of weights.

Selecting core subspaces. Continuing with the set $\mathbb{K}$ of candidate subspaces, this stage computes a smaller set $\mathbb{H} \subseteq \mathbb{K}$, as *core subspaces*, to cover most of the sparse preferences in $\mathcal{S}_s$. Note that the algorithm cannot simply choose the subspaces with the top weights because, together, they may overlap and end up covering only a small fraction of the preferences.

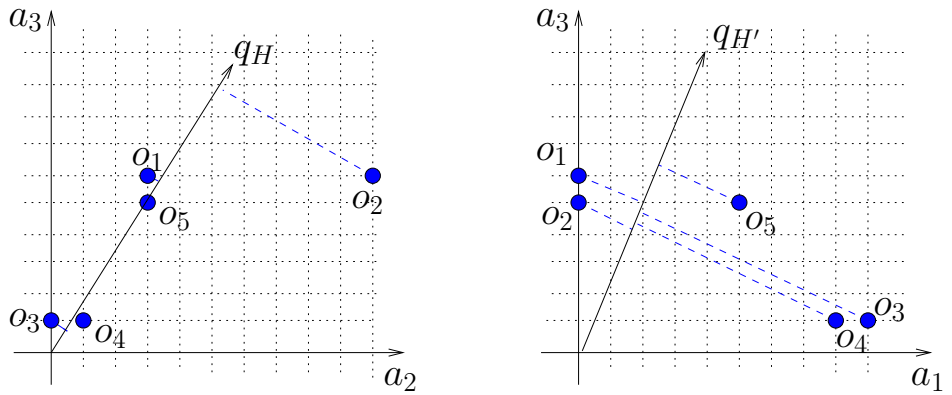
Algorithm 1 gives the pseudo-code of this approach. In each step, the subspace H with the highest weight is selected out from $\mathbb{K}$. Importantly, every time some H is picked, the set of preferences is "updated" in a way to reduce their contributions to subspace weights for those preferences covered by H. Thus, subsequent selections will focus on covering preferences that remain uncovered.

Algorithm 1: SelectCoreSubspaces($\mathbb{K}; \delta$).

$\mathbb{H} \leftarrow \emptyset$;
$m_s \leftarrow |\mathcal{S}_s|$; remember the original value for each $q \in \mathcal{S}_s$ (denoted $\tilde{q}$);
while $\frac{1}{m_s} \sum_{q \in \mathcal{S}_s} \|q\| \geq \delta$ **do**
    foreach $K \in \mathbb{K}$ **do** compute $w(K)$ using Eq. (4.1);
    $H \leftarrow \arg\max_{K \in \mathbb{K}} w(K)$;
    $\mathbb{H} \leftarrow \mathbb{H} \cup \{H\}$; $\mathbb{K} \leftarrow \mathbb{K} \backslash \{H\}$;
    foreach $q \in \mathcal{S}_s$ **do**
        $q \leftarrow q - \|\tilde{q}_H\| \cdot q_H$;
        if $\|q\| < \delta$ **then** $\mathcal{S}_s \leftarrow \mathcal{S}_s \setminus \{q\}$;
return $\mathbb{H}$;

If a preference q is contained in H, q is fully covered by H. Otherwise, q is only partially covered. In this case, q_H, the projection of q on H, provides information about some of the attributes of q in the sense that the ranking of objects w.r.t. q_H gives some information about ranking of objects w.r.t. q—for those attributes that are present in H. the algorithm reduces the weights of those attributes in q that are present in q_H, so that subspaces the algorithm selects in the future will capture the information of q w.r.t. the attributes of q not present in q. The simplest method will be to let $q \leftarrow q - q_H$; i.e., the algorithm simply clears q of any weights of attributes in H. However, this method is suboptimal; for a concrete example, see Figure 4.1.

Intuitively, for a partially covered preference, we would ideally like to cover each of its attributes with non-zero weights by multiple core subspaces. To this end, q is updated using $q \leftarrow q - \|\tilde{q}_H\| \cdot q_H$, where $\tilde{q}$ denotes the original vector for the preference (while q denotes the current vector, whose value changes over the course of the algorithm). The multiplier $\|\tilde{q}_H\|$ ensures that if $\tilde{q}$ is partially covered by H (i.e., $\|\tilde{q}_H\| < 1$), some residual weights will remain for attributes in H to encourage additional future coverage. On the other hand, if $\tilde{q}$ is contained in H, the vector will become zero after the update, and there is no need to consider q further. Consider the same example in Figure 4.1. After H has been selected, q will become $(0.2, 0.06, 0.1)$. Suppose $H' = (a_1, a_3)$ is chosen. As shown



(a) Subspace (a_2, a_3). (b) Subspace (a_1, a_3).

FIGURE 4.1: Illustration of coverage. Here, $d = 3$, $k = 2$, $q = (0.2, 0.3, 0.5)$, and $\mathcal{O} = \{o_1, \ldots, o_5\}$, where $o_1 = (0, 3, 6)$, $o_2 = (0, 10, 5)$, $o_3 = (9, 0, 1)$, $o_4 = (8, 1, 1)$, and $o_5 = (5, 3, 5)$. Thus, $\pi_{\leq 2}\langle q, \mathcal{O}\rangle = \{o_2, o_5\}$. Suppose $H = (a_2, a_3)$ is selected. Then $\pi_{\leq 2}\langle q_H, \mathcal{O}_H\rangle = \{o_2, o_1\}$, as shown in Figure 4.1(a). If the algorithm simply clears any weights of attributes in H, q becomes $(0.2, 0, 0)$ and the top 2 projected objects w.r.t. attribute a_1 are o_3 and o_4. In this case, the correct second-ranked object o_5 will not be reported.

in Figure 4.1(b), $\pi_{\leq 2}\langle q'_H, \mathcal{O}'_H\rangle = \{o_5, o_1\}$. Hence, the union of the top 2 objects in H and H', $\{o_1, o_2, o_5\}$, contains the exact top-2 objects, o_2 and o_5.

This idea of reducing weights slowly have been used in many different contexts, e.g., computing set covers of smaller sizes (see the survey [17]) than the standard greedy algorithm [30]. Section 4.6 presents experimental results that validate the effectiveness of multiple coverage in the context of this chapter.

In general, the algorithm stops covering a preference when its norm has dropped below a given significance threshold δ (e.g., 0.05). The algorithm stops selecting additional core subspaces altogether once the average norm of all preferences drops below δ.

Remarks. If Algorithm 1 tries to select too many core subspaces, it can simply terminate after reaching the desired number of core subspaces. In this case, the selected core subspaces may not be able to cover all sparse preferences. The uncovered preferences are handled using full-dimensional indexes (discussed in Section 4.4.3).

If $|\mathcal{S}_s|$ is large, instead of using the entire set to select core subspaces, the algorithm

can work with a subset of $\mathcal{S}_s$. Specifically, $\mathcal{S}_s$ is partitioned into buckets, such that within the same bucket, all preferences are "close"; e.g., for any two preferences q_i, q_j in the same bucket, $\langle q_i, q_j \rangle \geq \cos(\pi/6)$. Then, a random sample is chosen from each bucket and work with the samples to find core subspaces.

As mentioned in Section 4.1, the approach presented in this chapter can be seen as a generalization of the view-based approach [50, 68]. The indexes the algorithm builds for each core subspace H can be seen as a "super"-view that effectively provides the same power as materializing an infinite number of vector views whose vectors lie in H. On the other hand, unlike vector views, the core subspaces are axis-parallel. This restriction not only makes the problem more tractable, but also the attributes retain their meaning and if $\mathsf{Sp}(q)$ is a k-dim, then it will be k-dimensional even after the projection—number of non-zero attributes does not increase. It does not pose any issue for sparse preferences, because a multi-dimensional core subspace subsumes all vector views therein, including those that are not axis-parallel. Such degrees of freedom provided by multi-dimensional subspaces also make the core-subspace approach more robust—while the choices of vector views are susceptible to errors and changes in the distributions of attribute weight values in preferences, the core-subspace approach will still work well as long as preferences continue to specify non-zero weights, which can vary arbitrarily, for the same subsets of attributes.

4.4 Constructing Indexes

This section describes the indexes the presented algorithm builds. First, for each core subspace in $\mathbb{H}$, an object index is built for top-k queries (Section 4.4.1) and a preference index for reverse top-k queries (Section 4.4.2). The collection of these indexes for core subspaces aims at handling most (if not all) sparse preferences. Next, to handle all preferences not covered by these indexes, object and preference indexes are separately built for

the full-dimensional space (Section 4.4.3).

For reverse top-k queries, in addition to these indexes, the score of the k-th ranked object is also stored for each preference.

4.4.1 Core Subspace Indexes for Top-k Queries

For each core subspace $H \in \mathbb{H}$, a straightforward approach would to be project $\mathcal{O}$ onto H, and build an index on the $\dim(H)$-dimensional projected points that, given a query preference q, return the top k points with respect to q. This approach, however, has several issues. First, unless q is contained in H, there is a good chance that the some answers will be missed by looking only at the top k objects for q in H, even when the algorithm looks in multiple core subspaces partially covering q. Second, indexing all points in $\mathcal{O}$ for every core subspace results would require $O(n|\mathbb{H}|)$ space, which is too much. Third, looking in multiple core subspaces per query means that the index for each core subspace must be fast.

To address these issues, for each core subspace H, a small subset of objects is carefully chosen to build an index that supports top-βk queries in H. The index is small and fast, but approximate—a sensible trade-off because the top answers in a subspace in any case only approximate those in the full-dimensional space. Here, $\beta \geq 1$ is a small constant to increase the chance of catching a top-k object in the full-dimensional space. β is set to 3 in the experiments in Section 4.6; additional evaluation on the choice of β is presented in Section 4.6.3.

In more detail, let $\mathcal{O}_H$ denote the projection of $\mathcal{O}$, the set of input objects, on H, and let $\varepsilon > 0$ be the error allowance. An $(\beta k, \varepsilon)$-coreset of $\mathcal{C}_H \subseteq \mathcal{O}_H$ is constructed, as defined in Section 3.4.1. By definition, for any preference q in H, and for any $j \leq \beta k$, $\langle q, \pi_j(q, \mathcal{C}_H)\rangle \geq \langle q, \pi_j(q, \mathcal{O})\rangle - \varepsilon \bar{d}_j(q, \mathcal{O})$, i.e., the scores of top-βk objects of $\mathcal{C}_H$ are roughly the same as those of $\mathcal{O}$. In Chapter 3, we described an algorithm for computing coresets. Roughly speaking, it first applies an affine transformation to make $\mathcal{O}_H$ lie inside

a unit sphere centered at origin, and then proceeds in $2\beta k + 1$ passes. In each pass the algorithm carefully chooses a set $\mathcal{U}$ of $O(1/\varepsilon^{(\dim(H)-1)/2})$ points on a sphere of radius 2 centered at origin. For each point $u \in \mathcal{U}$, it computes an $(\varepsilon/2)$-approximate nearest neighbor of u in $\mathcal{O}_H$, say p_u. It adds the set $\{p_u \mid u \in \mathcal{U}\}$ to $\mathcal{C}_H$, removes it from $\mathcal{O}_H$, and proceeds with the next iteration. The details of how the points in $\mathcal{U}$ are chosen can be found in Chapter 3. In the worst case, $|\mathcal{C}_H| = O(\beta k/\varepsilon^{(\dim(H)-1)/2})$, but in practices the size of $\mathcal{C}_H$ is much smaller.

Next, an index is built on $\mathcal{C}_H$ such that for a query preference q in H and $\kappa \geq 1$, it returns $\pi_{\leq\kappa}(\mathcal{C}_H, q)$. By the definition of coreset, for $\kappa \leq \beta k$, the score of $\pi_{\leq\kappa}(\mathcal{C}_H, q)$ will be roughly the same as those of $\pi_{\leq\kappa}(\mathcal{O}_H, q)$. Many indexes are known for forward top-k queries; some provide provable bounds on their performance. Since this component is not the main focus of this chapter, the implementation simply uses $\dim(H)$ sorted lists on $\mathcal{C}_H$ and the TA algorithm for answering top-κ queries.

In the worst case, the total size of the index, summed over all core subspaces, is $O(\sum_{H\in\mathbb{H}} \beta k/\varepsilon^{(\dim(H)-1)/2})$. Since the dimensionality of core subspaces is capped at $\hat{\tau}$, the size is $O(\beta k|\mathbb{H}|/\varepsilon^{(\hat{\tau}-1)/2})$.

Remarks. Note that a preference workload influences the object indexes built in this section only through the choice of core subspaces. With a core subspace H, the object index is capable of handling any preference in H. This property makes the core-subspace approach more robust than the (vector) view-based approach [50, 68] with respect to errors and changes in the distributions of attribute weight values. Therefore, the core-subspace approach does not require a very detailed or accurate model of expected preference workload in order to support top-k queries effectively. Section 4.6 validate this observation experimentally.

4.4.2 *Core Subspace Indexes for Reverse Top-k*

Let $\mathcal{S}$ be the set of preferences with respect to which we wish to answer reverse top-k queries. On a high level, a small number of "covering core subspaces" is identified for each $q \in \mathcal{S}$. Then, the subset of preferences that H covers is indexed for each core subspace $H \in \mathbb{H}$. Before describing the indexes, we first discuss how to cover a preference.

Covering a preference with core subspaces. A *cover* of a preference q, denoted Γ_q, is a subset of $\mathbb{H}$, onto which the projections of q are intended to preserve the information about q, in the sense described in Section 4.3. A cover Γ_q is β-*perfect* with respect to $\mathcal{O}$ if for any query object $o \notin \mathcal{O}$ and $o \in \pi_{\leq k}(q, \mathcal{O} \cup \{o\})$, there exists a subspace $q \in \Gamma_q$ such that $o_H \in \pi_{\leq \beta k}(q_H, \mathcal{O}_H \cup \{o_H\})$. However, perfect covers are difficult to find. If q lies within a core subspace, then that subspace obviously is a 1-perfect cover of q. However, if none of the core subspaces contains q by itself, the best we can hope for is a small cover that preserves as much of q as possible.

A simple strategy would be to choose Γ_q to be those core subspaces that "overlap" with q (or more precisely, those on which q has a non-zero projection). However, there may be too many such subspace; picking them all would increase the index space and slow down queries. The top subspaces could be picked based on the norms of q's projections on them; however, doing so does not guarantee that all non-zero attribute weights of q are covered. Alternatively, the top subspaces could be picked according to their weights as defined in Section 4.3; however, weights are globally defined over $\mathcal{S}$ and not relevant for a particular q.

To avoid these problems, a limit ν is set on the maximum number of core subspaces in any cover, and use a greedy procedure (shown as Algorithm 2) to cover q. The algorithm is similar to Algorithm 1 in spirit (though now only one q is being covered). In each step, the algorithm always pick the core subspace H for which q_H has the largest norm. More

Algorithm 2: PreferenceCover$(\mathbb{H}, q; \nu, \theta)$

$\Gamma \leftarrow \emptyset$, $\tilde{q} \leftarrow q$;
while $\|q\| \geq \theta$ *and* $|\Gamma| < \nu$ **do**
  $H \leftarrow \arg\max_{H \in \mathbb{H}} \|q_H\|$;
  if $\|q_H\| = 0$ **then break**;
  $\Gamma \leftarrow \Gamma \cup \{H\}$; $\mathbb{H} \leftarrow \mathbb{H} \setminus \{H\}$;
  $q \leftarrow q - \|\tilde{q}_H\| \cdot q_H$;
if $\|q\| \geq \theta$ **then return** $\emptyset$;
return Γ;

importantly, the algorithm updates q for each step in a way that let subsequent picks focus on uncovered dimensions, while still encouraging multiple coverages for each dimensions (as discussed in Section 4.3). This process is repeated until q is "mostly covered," i.e., the residual norm is less than a given threshold θ, or the cover size exceeds the limit ν. The choices of ν and θ allow the trade-off between coverage completeness and cost. ν and θ are set to 3 and 0.5 in the experiments in Section 4.6, respectively; additional evaluation on their choices is presented in Section 4.6.3.

Remarks. Not all preferences can be covered. Algorithm 2 returns $\emptyset$ if it cannot cover a preference. It is even possible (though not very likely) that some sparse preference cannot be covered. On the other hand, it is also possible to cover a non-sparse preference. Preferences that cannot be covered will be handled separately by full-dimensional object and preference indexes (Section 4.4.3). The hope is that in practice, most preferences are sparse, can be covered, and will thus benefit from the core-subspace approach.

Building the preference index. For each core subspace H, let $\mathcal{S}^{(H)} = \{q_H \mid H \in \Gamma_q\}$ denote the subset of the preferences with H in their covers (as chosen by Algorithm 2). Given a query object $o \notin \mathcal{O}$, the goal is to build an index for finding all preference $q \in \mathcal{S}^{(H)}$ for which o_H ranks among the top βk objects in $\mathcal{O}_H \cup \{o_H\}$ for q_H. Assuming "near" β-perfect covers for all preferences, as discussed above, if o enters the top-k answer of any preference q, then q will be returned by querying the preference index of some core

subspace in Γ_q.

To build this preference index for $\mathcal{S}^{(H)}$, the algorithm considers, for each preference $q \in \mathcal{S}^{(H)}$, the score of the (βk)-th ranked object in $\mathcal{O}_H$ with respect to q_H, i.e., $\langle q_H, \pi_{\beta k}(q_H, \mathcal{O}_H)\rangle$. This score is called the *cutoff score*. Intuitively, it can be determined whether a query object o_H enters the top-k answer of q_H simply by comparing $\langle q_H, o_H\rangle$ with q_H's cutoff score. However, instead of working directly with $\mathcal{O}_H$, which is big, the algorithm works with $\mathcal{C}_H$, the $(\beta k, \varepsilon)$-coreset of $\mathcal{O}_H$ discussed in Section 4.4.1, which is much smaller. By definition, the score of the (βk)-th ranked object in $\mathcal{O}_H$ with respect to q_H is roughly the same as that of the (βk)-th ranked object in $\mathcal{C}_H$.

Let $\mathcal{S}_H = \{q_H \mid q \in \mathcal{S}^{(H)}\}$ denote the projection of $\mathcal{S}^{(H)}$ onto H. Indexing preferences in $\mathcal{S}_H$ and their cutoff scores in H is easier in the dual space (recall Section 4.2): the dual of $\mathcal{C}_H$ is a set $\mathcal{C}_H^*$ of hyperplanes in $\mathbb{R}^{\dim(H)}$, and each preference $q_H \in \mathcal{S}_H$ maps to a vertical ray q_H^*. In the following, it is assumed that preferences have positive weights for the last dimension of H (i.e., q_H^* is oriented toward the positive direction); the case of negative weights is analogous. Let $\mathcal{A}(\mathcal{C}_H)$ be the βk-level of $\mathcal{A}(\mathcal{C}_H)$, i.e., the query response surface for top-βk query. If the ray q_H^* intersects $\mathcal{A}(\mathcal{C}_H)$ at a hyperplane $o_H^* \in \mathcal{C}_H^*$, then $o_H = \pi_{\beta k}(q_H, \mathcal{C}_H)$. As discussed in Section 4.2, the intersection, denoted by χ_{q_H}, is the *cutoff point* of q_H. For any query object $z \notin \mathcal{C}_H$, $z \in \pi_{\leq \beta k}(q_H, \mathcal{C}_H \cup \{z\})$ iff the hyperplane dual to z lies below χ_{q_H}.

Let $\Xi_H = \{\chi_{q_H} \mid q_H \in \mathcal{S}_H\}$ denote the set of cutoff points for preferences in $\mathcal{S}_H$. An index is built for Ξ_H such that given a query hyperplane γ, it can report all points of Ξ_H lying above γ. As discussed in Section 4.2, there are several known indexes for this halfspace range query. The implementation is simply based on kd-tree, but with an additional optimization. Each node v of the index tree T is associated with a bounding box B_v and with $\Xi_v = B_v \cap \Xi_H$. In a standard kd-tree, as long as $|\Xi_v|$ is above some constant (node capacity), v is further split. The standard kd-tree construction algorithm is modified as follows. During construction, at each node v, the algorithm considers $\mathcal{C}_v^*$, the

subset of the hyperplanes in $\mathcal{C}_H^*$ that intersect B_v. If $|\mathcal{C}_v^*|$ is small and all cutoff points in Ξ_v lie within a small neighborhood, v is not split even if $|\Xi_v|$ is still above node capacity. Instead, v becomes a leaf and one cutoff point is chosen from Ξ_v to represent all of Ξ_v. Intuitively, in this case, if any query hyperplane lies above (or below) the representative cutoff point, then it will likely lie above (or below, resp.) all other cutoff points of Ξ_v. This optimization reduces the index size for highly clustered preferences.

Remarks. For a preference $q \in \mathcal{S}^{(H)}$, the closer $\|q_H\|$ is to $\|q\|$, the more likely it is for an object highly ranked w.r.t. q_H to also rank high w.r.t. q. Thus, instead of defining the cutoff point using always the (βk)-th ranked object w.r.t. q_H, it can be defined using the $(\beta' k)$-th ranked object, where $\beta' \in [1, \beta]$ is customized based on how close $\|q_H\|$ is to $\|q\|$. This heuristic expedites reverse top-k queries by tightening the cutoff condition; see Section 4.6.3 for more detailed discussion and evaluation.

4.4.3 Indexes for Uncovered Preferences

As discussed in Section 4.4.2, not all preferences are covered by the core subspaces. To handle such preferences, an object index (for forward top-k queries) and a preference index (for reverse top-k queries) are built in the full space $\mathbb{R}^d$.

Full-dimensional index for top-k queries. To make the index smaller and faster, instead of working with the entire set of objects $\mathcal{O}$, the algorithm works with a coreset (just like in Section 4.4.1, but now in the full d-dimensional space). A (k, ε)-coreset of size $O(k/\varepsilon^{(d-1)/2})$ can be computed using the algorithm described in [7]. Because of the exponential dependence on d, the coreset can be large even for moderate values of d. While it is known that this size is required for the worst case [7], as shown below, if the input objects lie on a low-dimensional algebraic surface of constant degree, then a smaller coreset can be computed.

Theorem 6. *Let $\mathcal{O}$ be a set of points in $\mathbb{R}^d$ that lie on a t-dimensional algebraic surface*

of constant degree, for $t < (d-1)/2$. Then, a (k,ε)-coreset of size $O((d^{3/2}/\varepsilon)^t)$ can be computed in time $d^{O(1)}n + O((d^{3/2}/\varepsilon)^t)$.

Proof. The algorithm by Agarwal et al. [7] is modified for computing a (k,ε)-coreset. Their algorithm works in $k+1$ phases, and in each phase computes a $(1,\varepsilon)$-coreset of size $O(1/\varepsilon^{(d-1)/2})$, using the technique in [5, 6]. The following technique from [5, 6] computes a $(1,\varepsilon)$-coreset of size of $O(1/\varepsilon^d)$: By applying an affine transform on $\mathcal{O}$ and its bounding box B, B is transformed to the hypercube $[0,1]^d$, so W.L.O.G. assume $B = [0,1]^d$. Draw a d-dimensional grid inside B so that the side length of each grid cell is at most $\varepsilon/\sqrt{d}$. Let $\mathbb{C}$ denote the set of resulting grid cells. $\mathbb{C}$ is induced by d families of hyperplanes, each consisting of $\lceil\sqrt{d}/\varepsilon\rceil$ hyperplanes. Let Γ be the set of these $O(d^{3/2}/\varepsilon)$ hyperplanes. For each cell $C \in \mathbb{C}$, if $C \cap \mathcal{O} \neq \emptyset$, choose one point of $C \cap \mathcal{O}$. It was shown in [5] that $\mathcal{C}$ is a $(1,\varepsilon)$-coreset of $\mathcal{O}$. Obviously, $|\mathcal{C}| = O((\sqrt{d}/\varepsilon)^d)$.

We prove an improved bound on $|\mathcal{C}|$ for our setting. Let Σ be a t-dimensional surface of constant degree that contains $\mathcal{O}$. That is, Σ is the common zero set of a family of $d-t$ d-variate polynomials, each of constant degree. We claim that Σ intersects $O((d^{3/2}/\varepsilon)^t)$ cells of $\mathbb{C}$; the constant proportionally depends on t as well as on the degree of Σ. Note that Σ can intersect only $O(1)$ cells without intersecting their boundaries, so it suffices to bound the number of cells of $\mathbb{C}$ whose boundaries intersect Σ. We prove this bound by induction on t.

For $t = 1$, Σ is a curve. It can intersect each hyperplane of Γ at $O(1)$ points, and therefore can intersect $O(|\Gamma|) = O(d^{3/2}/\varepsilon)$ cells of $\mathbb{C}$. For $t > 1$, fix a hyperplane $\gamma \in \Gamma$. The hyperplanes from the other $d-1$ families of Γ induce a $(d-1)$-dimensional grid $\mathbb{C}_\gamma$ on γ. Furthermore, $\gamma \cap \Sigma$ is a $(t-1)$-dimensional algebraic surface Σ_γ of constant degree. By induction hypothesis, Σ_γ intersects $O((d^{3/2}/\varepsilon)^{t-1})$ cells of $\mathbb{C}_\gamma$. Note that each $(d-1)$-dimensional face of a cell in $\mathbb{C}$ is a cell of $\mathbb{C}_\gamma$ for some $\gamma \in \Gamma$. Hence, summing over all hyperplanes of Γ, Σ intersects $O((d^{3/2}/\varepsilon)^t)$ cells of $\mathbb{C}$. Therefore, if Σ intersects

the boundary of a cell of $\mathbb{C}$, there exists a $\gamma \in \Gamma$ such that Σ_γ intersects a grid cell of $\mathbb{C}_\gamma$. Therefore, the size of the coreset is $O((d^{3/2}/\varepsilon)^t)$.

It is hard to compute the smallest box containing $\mathcal{O}$, but as observed in [5, 6], it suffices to compute a bounding box of $\mathcal{O}$ whose volume is within a constant factor of the minimum volume. Barequet and Har-Peled [24] described a simple $O(dn)$ time algorithm to compute such a box. We then repeat the above construction with this box. We omit some of the technical details here and conclude that $\mathcal{C}$ can be computed in $d^{O(1)}n + O((d^{3/2}/\varepsilon)^t)$ time. □

Full-dimensional index for reverse top-k queries. After computing the coreset $\mathcal{C}$ of $\mathcal{O}$, an index is built for the set $\bar{\mathcal{S}} \subseteq \mathcal{S}$ of uncovered preferences (i.e., those for which Algorithm 2 returns $\emptyset$). The procedure is the same as that described in Section 4.4.2 for indexing preferences for a core subspace, except that cutoff points are defined by the k-th ranked object instead of the (βk)-th.

4.5 Query Procedure

This section describes the procedure for answering top-k and reverse top-k queries using the indexes described in Section 4.4.

Top-k query. Given a query preference $q \in \mathbb{S}^{d-1}$, PreferenceCover$(\mathbb{H}, q)$ (Algorithm 2) is first called to compute Γ_q, a cover of q by core subspaces. There are two cases.

First, if $\Gamma_q = \emptyset$ (i.e., a cover of q cannot be found by $\mathbb{H}$), a query is issued to the full-dimensional object index described in Section 4.4.3 with q and $\pi_{\leq k}(q, \mathcal{C})$ is returned. Since $\mathcal{C}$ is a coreset of $\mathcal{O}$, the objects returned by the procedure approximate $\pi_{\leq k}(q, \mathcal{O})$.

Otherwise, $|\Gamma_q| > 0$ and q is covered. For each $H \in \Gamma_q$, q_H, the projection of q on H, is computed. A query is issued to the object index for H described in Section 4.4.1 in order to obtain the set of objects $\mathcal{S}_H \in \mathcal{O}$ corresponding to $\pi_{\leq \kappa_H}(q_H, \mathcal{C}_H)$, where $\kappa_H = k$

if $\|q_H\| \approx 1$, or $\kappa_H = \beta k$ otherwise. Then, $\pi_{\leq k}(q, \bigcup_{H \in \Gamma_q} \mathcal{S}_H)$, i.e., the top k objects among all returned objects, is computed by calculating their actual scores w.r.t. q.

Remarks. Note that more sophisticated methods for choosing κ are possible (see related discussion on flexible definition of cutoff points in the remarks at the end of Section 4.4.2). Intuitively, as $\|q_H\|$ increases, q_H becomes more like q, and a smaller κ_H (closer to k) will be enough to include top-k objects w.r.t. q with all high probability. The setting of $\kappa = k$ when $\|q_H\| \approx 1$ captures an important special case of this observation. See Section 4.6.3 for additional discussion and evaluation.

Reverse top-k query. Given a query object $o \in \mathbb{R}^d$, we want to report all affected preferences, i.e., any preference $q \in \mathcal{S}$ for which o is a top-k object in $\mathcal{O} \cup \{o\}$ w.r.t. q. First, affected preferences among the uncovered preferences $\bar{\mathcal{S}} \subseteq \mathcal{S}$ (i.e., those for which Algorithm 2 returns $\emptyset$) are found by querying the full-dimensional preference index on $\bar{\mathcal{S}}$ described in Section 4.4.3.

Next, affected preferences are found among the covered preferences, $\mathcal{S} \setminus \bar{\mathcal{S}}$. For each subspace $H \in \mathbb{H}$, we determine whether o is "relevant" to H, in the sense whether there can be some preference q in H for which o_H is potentially one of the top-βk objects of $\mathcal{C}_H \cup \{o_H\}$ w.r.t. q. The procedure for testing relevance is given in Chapter 3; it takes $O(k/\varepsilon^{(d-1)/2})$ time in the worst case. If o is relevant to H, a query is issued to the preference index with o for H described in Section 4.4.2 in order to find the affected preferences in H with their cutoff points. For each such preference q found, o's actual score w.r.t. q in the full space is further calculated, and q is returned only if o's score is higher than q's k-th score that is stored (as discussed at the beginning of Section 4.4).

Remarks. In the worst case, a query object o may be relevant to all core subspaces, but in practice, o is often relevant to only a few core subspaces, so the relevance test is useful.

4.6 Experimental Evaluation

Approaches compared. This section compares the core-subspace approach, hereafter referred to as ***CSI*** (for ***C**ore-**S**ubspace-based **I**ndexing*), with a number of alternatives. All approaches are implemented in C++.

For top-k queries, the following alternatives are considered. ***Scan*** is a brute-force method that examines all objects. ***BB*** indexes all objects in a d-dim kd-tree and uses a branch-and-bound algorithm to search for the top k objects. ***TA***, the Threshold Algorithm, keeps a list of objects sorted by each attribute; to find the top k objects give a query preference q, it uses the lists for attributes with non-zero weights specified by q. ***PCA+TA*** first applies PCA (principal component analysis) to reduce the dimensionality of the objects, and then uses *TA*. ***Views***, the view-based approach, randomly selects as views a set of unit vectors from a given preference distribution, and materializes their top βk objects. Given a query preference q, it retrieves the top βk objects from ν views most similar to q and computes the top k among these objects.

For reverse top-k queries, all approaches store the score of the k-ranked object for each preference. ***Scan*** examines all preferences. ***HSR***, for *halfspace range search*, answers the query in the dual space using a d-dim kd-tree on the cutoff points, as described in Section 4.2. ***PCA+HSR*** first applies PCA and then uses *HSR* in the reduced space. **Views** selects its views as described above, and assigns each preference to ν views; given a query object o, it retrieves all preferences assigned to views for which o enters their top-βk list, and filters these preferences to find those affected by o.

Since *CSI* is approximate, ε is set to 0.08 to be the error allowance, such that coresets are sized to provide answers whose scores are within ε times the directional width of the objects with respect to a query preference (recall Eq. (3.1)). To ensure fair comparison between *CSI* and *views*, the same settings of $\beta = 3$ and $\nu = 3$ are used, and the number of views is chosen such that the total space consumption of *views* is the same as that of *CSI*.

Performance metrics. For a given query workload, the average wall-clock time per query over the workload is reported, as measured on a Dell OptiPlex 990 with 3.40GHz Intel Core i7-2600 CPU, 8MB cache, and 8GB memory.

For approximate approaches to top-k queries (*CSI*, *PCA+TA*, and *views*), the approximation error is measured for each query object o as follows. Let $\tilde{o}_i$ denote the i-th ranked object returned by an algorithm. The error is computed as $\max_{i \in [1,k]} \frac{\langle q, \pi_i(q,\mathcal{O})\rangle - \langle q, \tilde{o}_i\rangle}{\varepsilon \bar{d}_i(q,\mathcal{O})}$, where ε is the error allowance as set above. Thus, an error of 1 or less is considered "acceptable." The RMS (root mean square) error over the query workload is reported. If RMS error is 1 or higher, it is likely that a significant fraction of the errors are unacceptable.

For approximate approaches to reverse top-k queries (*CSI*, *PCA+HSR*, and *views*), their approximate qualities are measured using *false negative rates* defined as follows. Given a query object o, a preference q is considered to be *significantly* affected by o iff $\langle q, o\rangle > \langle q, \pi_k(q,\mathcal{O})\rangle + \varepsilon \bar{d}_k(q,\mathcal{O})$; here, the same ε we set earlier defines the amount of acceptable slack. If a significantly affected q is missing from the result query result, it is counted as a false negative. The total number of false negatives is divided by the total actual number of significantly affected preferences over the entire query workload, and this ratio is reported as the false negative rate.

Synthetic object workloads. Objects are generated using a number of distributions. With ***box-uniform***, objects are distributed uniformly and randomly *within* the unit box in $\mathbb{R}^d$. With ***sphere-uniform***, objects are distributed uniformly and randomly *on the surface of* the unit sphere in $\mathbb{R}^d$. With ***sector-select***, objects are drawn randomly from a spherical cap in $\mathbb{R}^d$ with apex at the origin, and with radius 1 and cone angle $15°$; furthermore, an object is generated if it ranks high w.r.t. some preference in the preference workload. With ***t-surface***, objects lie on a t-dimensional algebraic surface embedded in the original space. The surface is defined using t parameters. For each attribute, a multivariate polynomial of constant degree is defined from the t parameters. To generate an object, values are

first generated for the t parameters and then the object coordinates are computed using the polynomials.

Synthetic preference workloads. The preference workload generator uses a number of parameters to control workload characteristics. Given a ***fraction of non-sparse preferences***, this fraction of the preferences is generated in the workload by picking unit vectors in $\mathbb{R}^d$ uniformly at random; assuming a sufficiently large d, such preferences are almost always non-sparse. The remaining (sparse) preferences are generated from a set $\mathbb{G}$ of "generating subspaces," where $|\mathbb{G}| = h_{\text{gen}}$, the ***number of generating subspaces***, and for each $G \in \mathbb{G}$, $\dim(G) \leq \tau_{\text{gen}}$, the ***maximum generating density***. $\mathbb{G}$ is picked in two ways: with ***uniform generating subspaces***, every subspace with dimensionality no more than τ_{gen} has an equal probability of being picked; with ***skewed generating subspaces***, each attribute is assigned a popularity, such that popular attributes are more likely to be included in a generating subspace. To generate a preference, a generating subspace $G \in \mathbb{G}$ is selected at random. Then, the preference is generated in two ways: with ***uniform preferences within subspaces***, a unit vector in G is uniformly drawn at random; with ***clustered preferences within subspaces***, preferences are drawn from a mixture distribution centered around a small number of randomly chosen unit vectors in G.

NBA workload. The dataset contains 17 career stats for $3{,}861$ NBA players. Preferences are still generated synthetically.

Document subscription workload. This workload is intended to approximate an application scenario where users subscribe to documents of their interest. The set of objects is generated to represent documents from the collection of approximately $300{,}000$ NY Times news articles [19]. A singular value decomposition (SVD) is performed on the documents to discover the underlying 20 most relevant topics. Hence, each document is mapped a

point in the 20-dimensional space, where each attribute represents a topic.

Next, the Yahoo! search query collection [118] is used to extract the set of preferences for this workload. This collection contains a random sample of $4{,}496$ queries posted to Yahoo!'s US search engine in January, 2009. The queries are preprocessed to discard stop words and words that are not present in the document collection. Then, using the same SVD matrices, each query is mapped to a unit vector in the 20-dimensional space; if a component of the vector is below a threshold t, it is set to 0. The table below shows, for two different t values, the density (number of non-zero components) distribution of resulting vectors (recall that $d = 20$):

density	0	1	2	3	4	5	6	7	8	9	10	11	12
# vectors ($t = 0.05$)	1558	0	0	3	32	113	342	749	864	567	209	51	8
# vectors ($t = 0.1$)	1592	338	1010	1084	390	80	2	0	0	0	0	0	0

In the experiments, t is set to 0.1. To get a larger set of preferences, they are generated from the above set of "seed" vectors. Each word is associated with its 5 most "probable" topics (derived from the same SVD). Let two words be neighbors if they are associated with a common topic. Starting from a seed vector, new preferences are generated by iteratively replacing one of its words with a neighboring word.

4.6.1 Top-k Query Performance

Varying the fraction of non-sparse preferences. We begin by studying the effect of the fraction of non-sparse preferences on top-k queries for various approaches. Here, $d = 80$, $k = 5$, and $100{,}000$ objects are generated from *box-uniform*. The query workload consists of $10{,}000$ preferences; the sparse ones among them are *uniform preferences* drawn from 200 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. *CSI* and *views* are given $10{,}000$ preferences generated from the same distribution in constructing their indexes. In Figure 4.2, the fraction of non-sparse preferences varies from 0 to 0.8. For *CSI*, the RMS error is comfortably below 1 at all times, but the overall average query

time rises with more non-sparse preferences. The table below shows the fraction of query preferences that are covered by core subspaces, which has a roughly linear relationship with the fraction of sparse preference:

Fraction of non-sparse preferences	0.0	0.2	0.4	0.6	0.8
Fraction of covered queries	98.9%	78.2%	56.4%	32.0%	7.90%

Recall that *CSI* uses indexes in core subspaces for covered preferences, and the full-dimensional coreset for uncovered preferences. To better see their performance difference, *CSI** is used in this and following figures to show the average query time for covered preferences. When most preferences are non-sparse, they are handled by the full-dimensional coreset, so *CSI* becomes as slow as *scan* and *BB*,[2] which is expected in high dimensions. This observation implies that using only the full-dimensional coreset (as well as other full-dimensional approaches such as the layer-based ones mentioned in Section 4.1) will not work in high dimensions.

The error of *views* is acceptable when all preference are sparse. However, its error quickly deteriorates as the fraction of non-sparse preference rises, because of the inherent difficulty in capturing high-dimensional space with vector-based views. Although the query-time plot shows an apparent advantage of *views* over *CSI* when the fraction of non-sparse views is at least 0.2, this advantage is not real—to make its error acceptable, *views* would have to use a lot more views, driving the space and query time higher than *CSI*.

Figure 4.2 shows that *PCA+TA* does not produce acceptable errors; thus, its query time is not plotted. Also, error for *scan*, *BB*, and *TA* are not plotted because they are exact methods.

Now that the effect of non-sparse preferences is well understood, this section will focus on workloads where all preferences are sparse—extrapolation to the general case is easy, and *views* will only be worse than *CSI* with more non-sparse preferences.

[2] *TA* is slower with more non-sparse preferences, because each such preference requires processing d lists and is thus more costly than a sparse one.

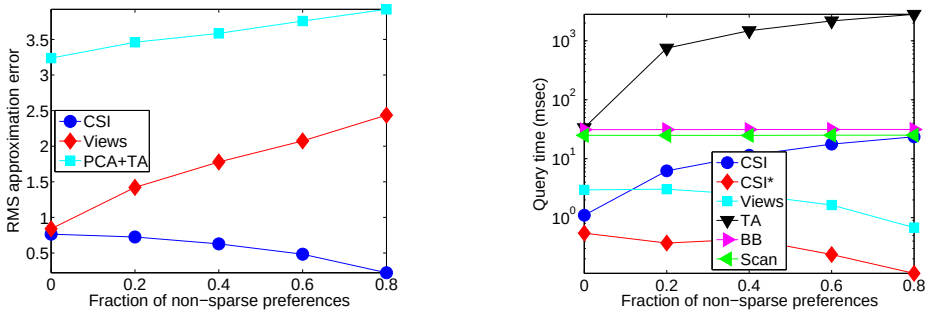


FIGURE 4.2: Top-k queries when varying the fraction of non-sparse preferences; $d = 80$, $n = 100{,}000$, $k = 5$.

Varying preference workloads. We now examine several different preference workloads. In Figure 4.3, $d = 20$, and preferences (either for querying or for index construction) are generated from *uniform generating subspaces*; in Figure 4.4, $d = 80$, and generating subspaces are *skewed*. For both figures, the number of generating subspaces varies from 50 to 500. Other workload parameters remain the same as Figure 4.2. The main observation is that the exact methods run much slower than the approximate ones (note the logarithmic scale of the query time axis). *CSI* and *views* and have comparable query time, but *CSI* has smaller errors than *views*. *PCA+TA* again produces much higher errors than *CSI* and *views*.

Going from Figure 4.3 to Figure 4.4, queries generally become slower with a higher dimensionality, but as indicated by *CSI**, query times for covered preferences remain short, and become much shorter than *views*. As majority of the queries are covered, they will benefit from shorter-than-average query times. On the other hand, the accuracy lead of *CSI* over *views* is consistent in both Figures 4.3 and 4.4.

The number of generating subspaces has some effect on performance, though this effect is not strong enough to change any conclusion in our discussion above.

Varying dimensionality. Next, let's consider the impact of dimensionality. Again, $k = 5$, and $100{,}000$ objects are generated from *box-uniform*. Preferences are drawn as *uniform*

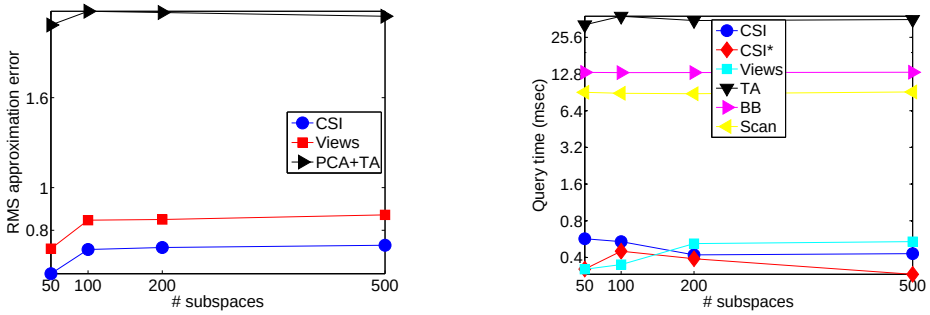


FIGURE 4.3: Top-k queries when varying the number of *uniform* generating subspaces; $d = 20$, $n = 100{,}000$, $k = 5$.

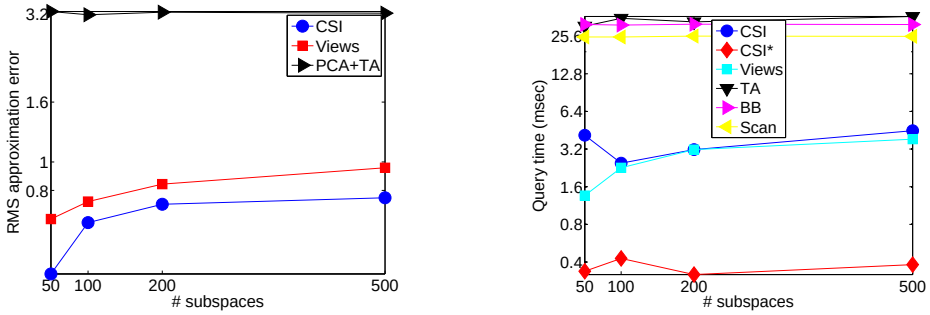


FIGURE 4.4: Top-k queries when varying the number of *skewed* generating subspaces; $d = 80$, $n = 100{,}000$, $k = 5$.

preferences from 100 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. Figure 4.5 shows that as *CSI* consistently delivers higher accuracy than *views* across all dimensionalities, and its big lead over *PCA+TA* widens as d increases. While *views* starts out to be faster than *CSI* in low dimensions, the speed gap between quickly narrows in higher dimensions. The exact methods are generally much slower *CSI* and *views*. Finally, looking at *CSI**, we see that covered queries remain extremely fast despite the increase in d, meaning that core subspaces do a good job of protecting sparse preference query performance from the curse of dimensionality.

Objects from low-dimensional algebraic surfaces. In this experiment, a varying number of objects is drawn from *t-surface* (a 3-dimensional bounded-degree algebraic surface to be specific). Here, $d = 100$, and preference workloads are generated by drawing $10{,}000$

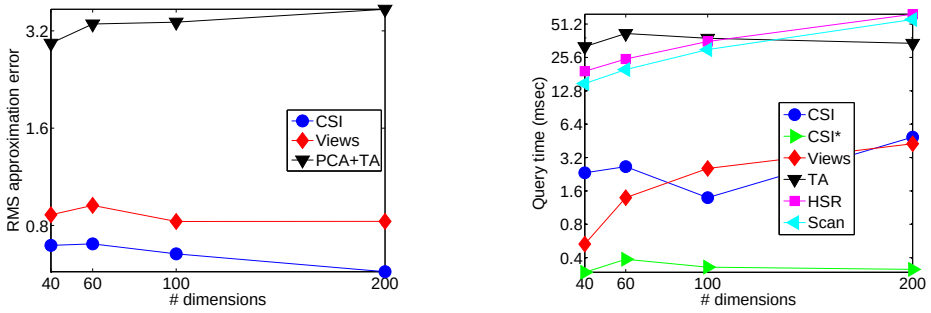


FIGURE 4.5: Top-k queries when varying d; $n = 100{,}000$, $k = 5$.

uniform preferences from 100 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. Figure 4.6 shows that *CSI*'s query time (which accounts for uncovered query preferences that will use the full-dimensional coreset) remains steady as the number of objects increases. In fact, despite high dimensionality ($d = 100$), the size of *CSI*'s full-dimensional coreset is only around $6{,}600$ even when $n = 200{,}000$, confirming the effectiveness of the improvement to the coreset construction algorithm discussed in Section 4.4.3. In comparison, the exact methods are much slower, and the gap widens as n increases. *Views* is also slower than *CSI*, but the gap does not widen thanks to *CSI*'s small coreset size (recall that the space of *views* is set to be the same as that of *CSI*).

Figure 4.6 also shows an approximate variant of *TA* called *ApproxTA*, which simply runs *TA* on the full-dimensional coreset used by *CSI*, for all query preferences. Between *ApproxTA* and *CSI*, there is a clear tradeoff—*ApproxTA* has better accuracy, while *CSI* has faster speed. This comparison highlights the benefit of the improved corset construction algorithm, as well as the ability for core subspace to further provide good accuracy/speed trade-offs.

Sensitivity to changes in preference distribution. Section 4.4.1 argued that *CSI* is more robust than *views* with respect to errors and changes in the distributions of attribute weight values. This claim is now validated using the following experiment. Here, $d = 80$, $k = 5$, and we use $100{,}000$ objects from *sector-select*. Two preference workload distributions W_1

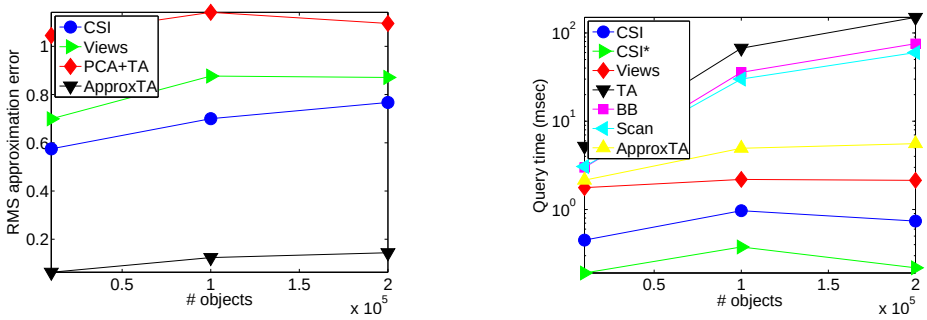


FIGURE 4.6: Top-k queries for objects from *t-surface*; $d = 100$, $k = 5$.

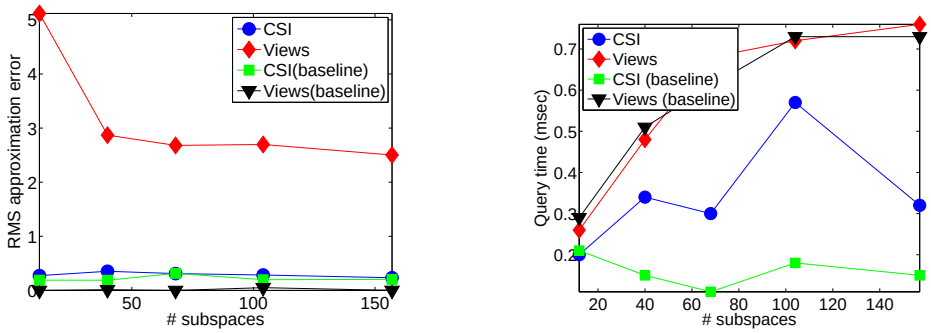


FIGURE 4.7: Sensitivity of top-k query performance to changes in preference distribution within generating subspaces; $d = 80$, $n = 100{,}000$, $k = 5$.

and W_2 are defined. Both use *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 3$; we also use these subspaces to generate the sectors for *sector-select* objects. W_1 and W_2 both draw *clustered preferences* from each generating subspace, but they have different set of cluster centers. To construct their indexes, both *CSI* and *views* are given $10{,}000$ preferences from W_1. Then, the performance of *CSI* and *views* are compared when given $10{,}000$ query preferences from W_1 (i.e., preference distribution is *unchanged*) and when given query $10{,}000$ preferences from W_2 (i.e., preference distribution is *changed*).

Figure 4.7 plots the results when varying the number of generating subspaces; results for which the preference distribution is unchanged are shown as "baseline." THe figure shows that while *views* has a very accurate baseline (because the preferences are highly clustered), its accuracy simply becomes unacceptable when the preference distribution changes. In contrast, *CSI* remains highly accurate despite the change.

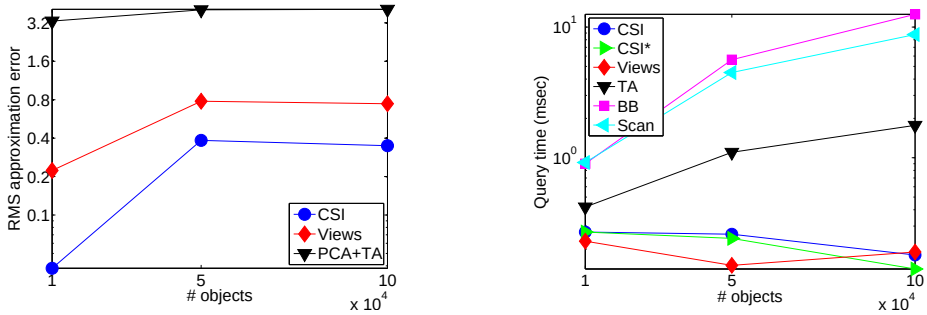


FIGURE 4.8: Top-k queries for document subscription workload; $d = 20$, $k = 5$.

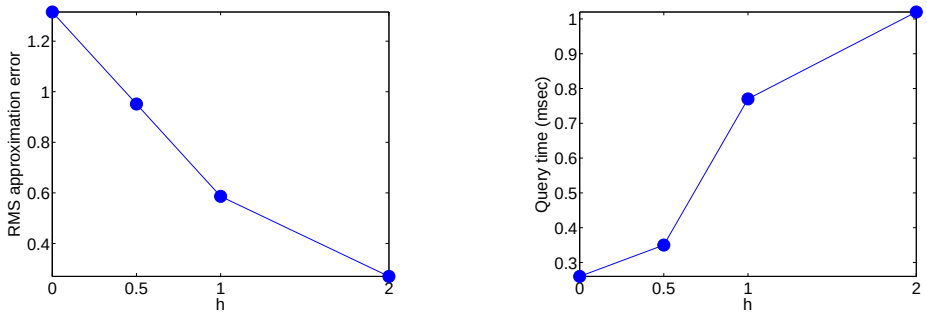


FIGURE 4.9: Top-k queries when varying incentive for multiple coverage.

Document subscription workload. Figures 4.8 shows the results for the document subscription workload when varying the number of documents. Once again, the results confirm the effectiveness of *CSI*. Almost 100% of the query preferences can be handled by core subspaces, and the average query time is much faster than the exact methods and comparable with *views*. In comparison, *views* has bigger approximation errors, and *PCA+HSR* is worse. In fact, under *CSI*, at most 3% of the queries exceed the prescribed error allowance (i.e., approximation error is greater than 1). In contrast, up to 8% and 84% of preferences have approximation errors greater than 1 under *views* and *PCA+HSR*, respectively.

Benefit of multiple coverage. Here, the effectiveness of multiple coverage is tested for the box-uniform workload when varying incentive for multiple coverage. In the setting, $d = 80$, $k = 5$, and $10{,}000$ *uniform preferences* from 200 *uniform generating subspaces* with

maximum dimensionality $\tau_{\text{gen}} = 6$. Recall that when H is selected, for each preference q, the presented algorithm reduces the weights of those attributes in q that are present in q_H. For single coverage, $q \leftarrow q - q_H$. For multiple coverage, $q \leftarrow q - \|\tilde{q}_H\| q_H$. In Figure 4.9, h indicates the incentive for multiple coverage, i.e., q is updated using $q \leftarrow q - \|\tilde{q}_H\|^h q_H$. The figure shows that if we simply clear q of any weights of attributes in H ($h = 0$), the approximation error is greater than 1. As h increases, the error decreases but the query time goes up. By setting $h = 1$, good balance is acheived between approximation error and query time.

4.6.2 Reverse Top-k Query Performance

Varying dimensionality. We begin by studying the effect of dimensionality on reverse top-k queries for various approaches. Here, $k = 5$. $2{,}000$ objects are drawn from *box-uniform*, and $100{,}000$ *uniform preferences* from 100 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. Query objects are also drawn from *box-uniform*. Figure 4.10 shows the results. As with top-k queries, a similar pattern is found in accuracy: *CSI* misses very few significantly affected preferences (no more than about 4%); *views* misses 18% to 45% as d increases; *PCA+HSR* misses over 90%. In terms of query time, *scan* is the slowest, as expected. *HSR* is reasonably fast as an exact method in low dimensions; however, its lead over *scan* narrows quickly as d increases—a query halfspace intersects more nodes of the underlying kd-tree, and the cost of determining whether a cut-off point lies above a hyperplane grows proportionally. Among the approximate methods, both *PCA+HSR* and *views* are faster than *CSI*, but they have poor accuracy. Keeping the accuracy high, *CSI* still manages to offer a significant speedup over *scan* even at $d = 200$.

Recall that for each query, *CSI* also checks the full-dimensional index of uncovered preferences, basically using *HSR*. This cost component is reflected in the reported query times, and depends on the fraction of the uncovered preferences. In the worst case, if all preferences are non-sparse, many of them will not be covered, and the query time of

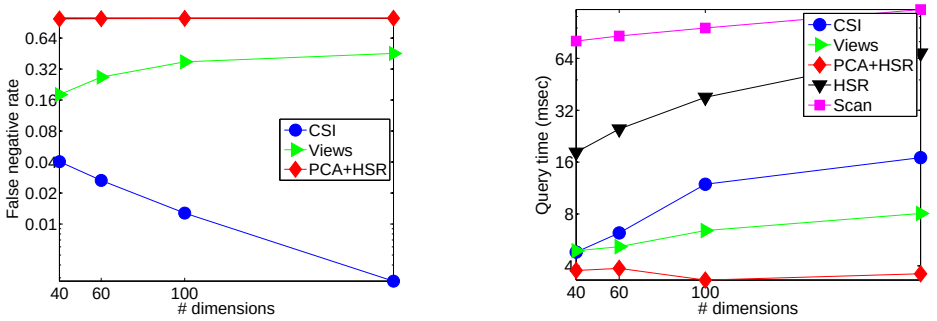


FIGURE 4.10: Reverse top-k queries when varying d; $n = 2{,}000$, $m = 10{,}000$, $k = 5$.

CSI will be similar to that of *HSR*. Because it is easy to extrapolate the effect of varying the fraction of non-sparse preferences, this fraction is set to 0 and do not vary it for the synthetic workloads in this section.

Varying the number of generating subspaces. The same workload parameters are used as in Figure 4.10, but vary the number of generating subspaces while fixing $d = 40$. Figure 4.11 shows the results. Again, a similar trade-off is shown as in Figure 4.10: *views* and *PCA+HSR* run faster than *CSI*, but offer much lower accuracy; the exact methods are much slower.

Figure 4.11 shows that the number of generating subspaces has an impact on *CSI*. More generating subspaces imply more diversity in preferences, which leads to more core subspaces (16 core subspaces for 50 generating subspaces vs. 25 for 500), as well as a larger number of imperfectly covered preferences. Hence, both false positive rate and query time increase, although the effect is not strong enough to change any conclusion in our discussion above.

Varying the number of preferences. Next, we study the effect of the number of preferences. The same preference workload parameters are used as in Figure 4.10, but vary the number of preferences up to $500{,}000$. This time, the $2{,}000$ objects are from *sphere-uniform*, and query objects are also drawn from *sphere-uniform*. Figure 4.12 shows that

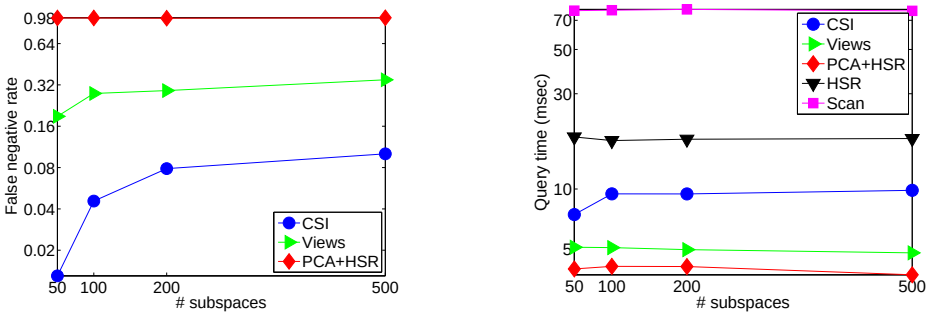


FIGURE 4.11: Reverse top-k queries when varying the number of generating subspaces; $d = 40$, $n = 2{,}000$, $m = 100{,}000$, $k = 5$.

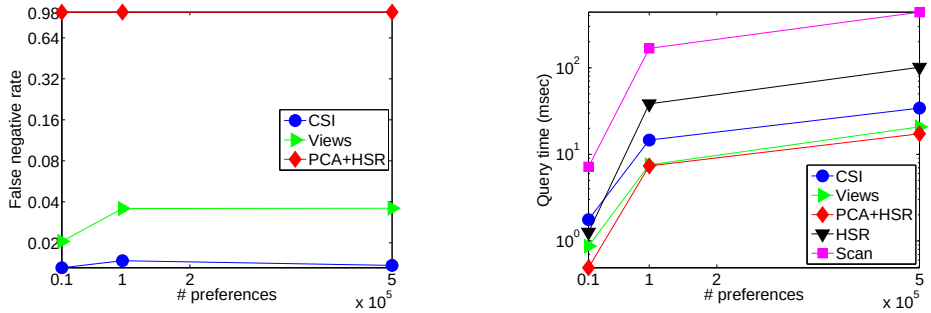


FIGURE 4.12: Reverse top-k queries when increasing m; $d = 40$, $n = 2{,}000$, $k = 5$.

the same trade-off identified in previous figures continues: *CSI* is slower than *views* and *PCA+HSR*, but is more accurate. The exact methods are much slower, while the fastest approximate method, *PCA+HSR*, misses most of the answers.

Overall, *CSI* demonstrates good scalability in the number of preferences. With half a million preferences, *CSI*'s false negative rate is merely 1.4%, and average query time is under 35 milliseconds. A more detailed breakdown shows that it spends 15.14ms querying indexes for core subspaces, and 16.15ms filtering false positives; it also spends 2.89ms on checking the full-dimensional index for $19{,}678$ uncovered preferences (out of $500{,}000$). In comparison, the average query time of *views* is about 21 milliseconds, 91% of which is spent on filtering false positives.

NBA workload. Figures 4.13 compares various approaches for the NBA workload as the number of preferences increases. *Uniform preferences* are drawn from 100 *uniform*

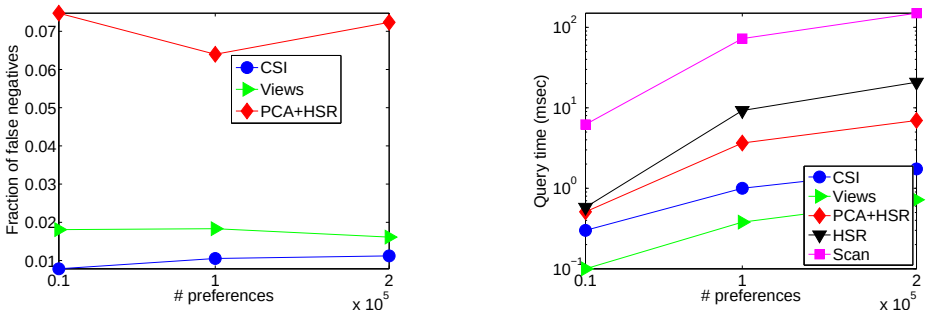


FIGURE 4.13: Reverse top-k queries for NBA workload; $d = 17$, $n = 3{,}861$, $k = 5$.

generating subspaces with maximum dimensionality $\tau_{\text{gen}} = 6$. To ensure that the query objects are "interesting" (i.e., likely affecting some preferences), the reverse top-k queries are tested using Hall-of-Fame players as query objects. Figure 4.13 shows that *views* has the fastest query time across all tested workloads (beating *PCA+HSR*), but *CSI* achieves the lowest false negative rate among all approximate methods, while still delivering fast query time with a large number of preferences.

Document subscription workload. For this workload, $2{,}000$ documents are used and the number of preferences varies up to $200{,}000$. Figure 4.14 shows the results. As with the NBA workload, both *views* and *CSI* perform well; additionally, the exact method *HSR* also has acceptable query time in this case. *CSI* offers a nice middle ground between *HSR* and *views*: on one hand, *CSI* is 2 times faster than *HSR*; on the other hand, it is 2 to 3 times slower than *views*, but its false negative rate is 30% to 50% lower than *views*. For *CSI*, the false negatives rate is less than 1% across all tested workloads.

Benefit of multiple coverage. Figure 4.15 shows the results for the box-uniform workload when varying incentive for multiple coverage. In the setting, $d = 80$, $k = 5$, and $10{,}000$ *uniform preferences* from 200 *uniform generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. Again, h indicates the incentive for multiple coverage, i.e., q is updated using $q \leftarrow q - \|\tilde{q}_H\|^h q_H$. The figure shows that if we simply clear q of any weights of attributes

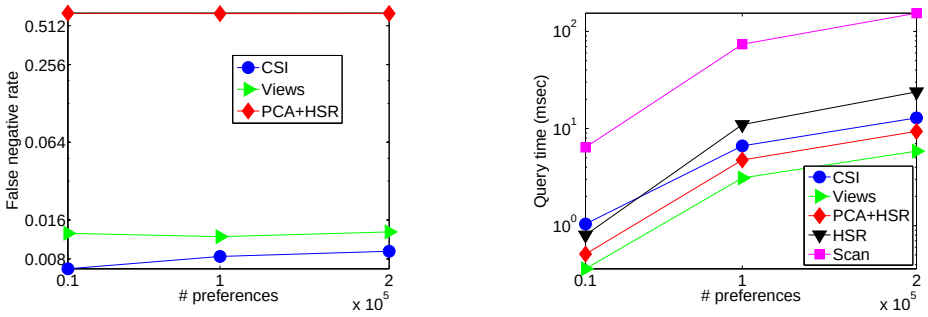


FIGURE 4.14: Reverse top-k queries for document subscription workload; $d = 20$, $n = 2{,}000$, $k = 5$.

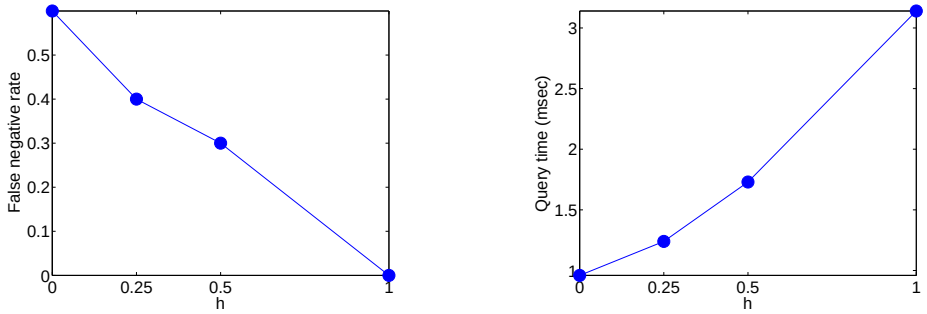


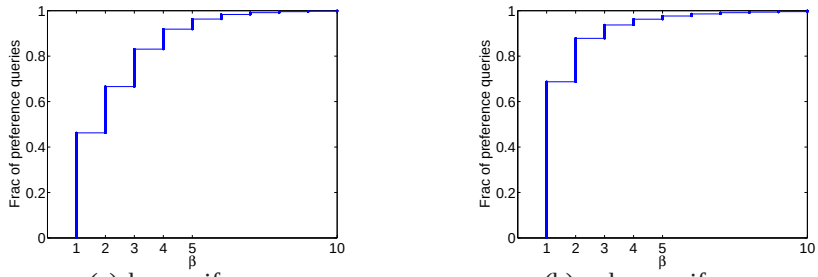
FIGURE 4.15: Reverse top-k queries when varying incentive for multiple coverage.

in H ($h = 0$), the false negative rate is above 0.5. If we enourage additional coverage for those weights, the false negative rate decreases but the query time goes up. By setting $h = 1$, the false negative rate is less close to 0.

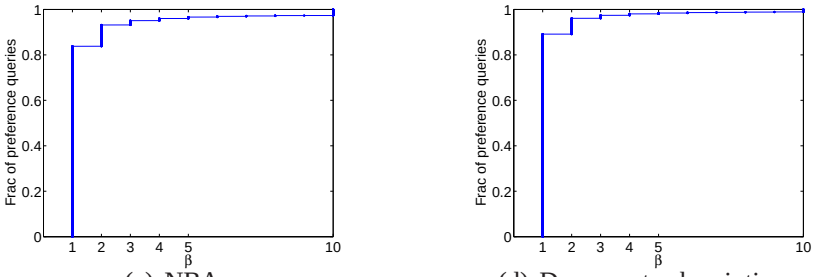
4.6.3 Algorithm parameters

Different choices of parameters for *CSI* have been experimented to verify the settings of parameters.

Parameter β. The smallest value for β is estimated across different workloads s.t. most preferences are within the prescribed error allowance if top-βk objects are retreived from each core subspace in their preference covers. Here, $\varepsilon = 0.08$ and $k = 5$. Unless specified, $10{,}000$ preferences are drawn from 200 uniform *generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$.



(a) box-uniform (b) sphere-uniform



(c) NBA (d) Document subscription

FIGURE 4.16: Parameter β

The value of β is estimated as follows. A preference is said to cover Γ_q is β-*approximate* if $\max_{i\in[1,k]} \frac{\langle q,\pi_i(q,\mathcal{O})\rangle - \langle q,\pi_i(q,\mathcal{O}')\rangle}{\varepsilon \overline{d}_i(q,\mathcal{O})} \leq 1$, where $\mathcal{O}'$ is the union of top-βk objects of all subspaces $H \in \Gamma_q$. For each preference q, its cover Γ_q is first computed. If q can be covered with at most ν core subspaces, i.e., $|\Gamma_q| \leq \nu$, the smallest integer β is computed s.t. Γ_q is β-approximate.

Figure 4.16(a) shows the cumulative distribution function (cdf) of β for the *box-uniform* workload, which contains $100{,}000$ objects with $d = 20$. The figure shows that 80% and 90% of preference covers are 3-approximate and 4-approximate, respectively. Figure 4.16(b) shows the cdf of β for $100{,}000$ objects drawn from the *sphere-uniform* distribution with $d = 80$. Roughly 90% and 95% of preference covers are 2-approximate and 3-approximate, respectively. For the NBA workload, $d = 17$ and $|\mathcal{O}| = 3861$. As shown in Fig. 4.16(c), roughly 80% of preference covers are 1-approximate. Figure 4.16(d) shows the results for the document subscription workload, which contains $100{,}000$ documents and $10{,}000$ document subscriptions with $d = 20$. About 90% of preference covers are 1-approximate, and almost all preference queries are within the prescribed error allowance

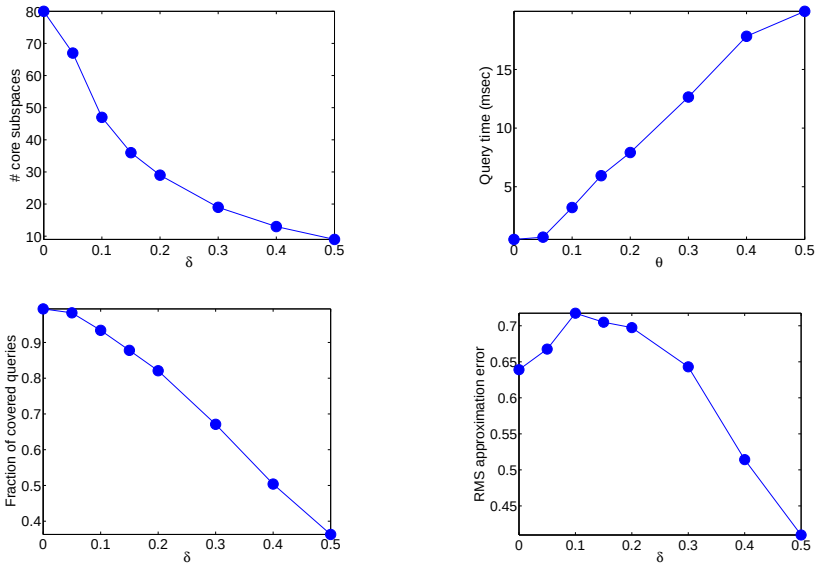


FIGURE 4.17: Vary δ in Algorithm 1; $\theta = 0.5$ in Algorithm 2.

when β is set to 2.

For each $q \in \mathcal{S}$, let $H^* \in \Gamma_H$ denote the subspace in Γ_H that covers most weights of q among all subspaces $H \in \Gamma_H$, i.e., $\|q_{H^*}\| \geq \|q_H\|$ for all $H \in \Gamma_q$. Recall that for a reverse top-k queries, instead of defining the cutoff point using always the (βk)-th ranked object w.r.t. q_H, the cutoff condition can be tightened by defining it using the $(\beta' k)$-th ranked

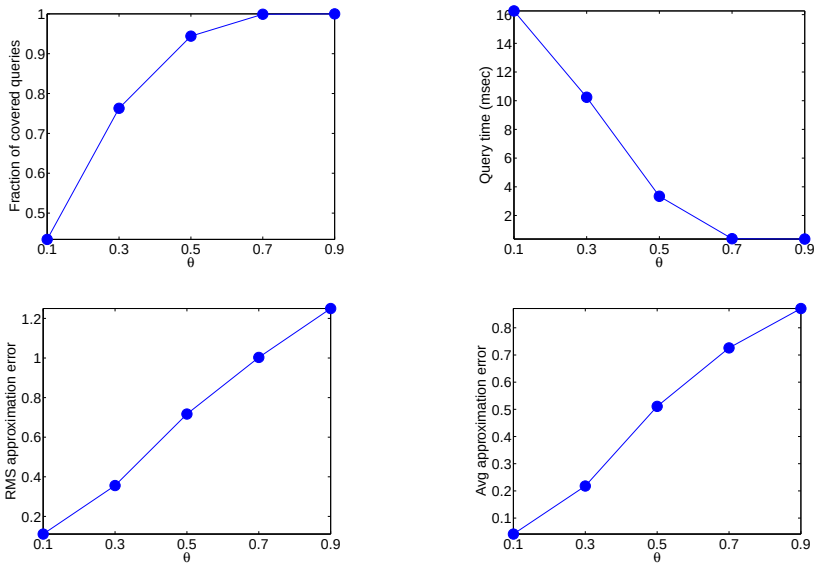


FIGURE 4.18: Vary θ in Algorithm 2; $\delta = 0.1$ in Algorithm 1.

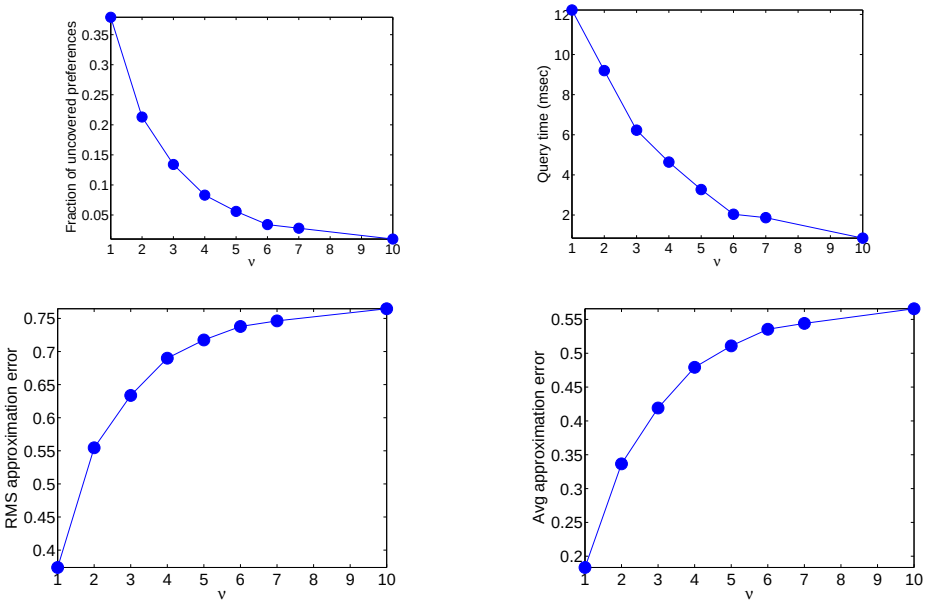


FIGURE 4.19: Vary ν in Algorithm 1; $\theta = 0.5$ in Algorithm 2.

object, where $\beta' \in [1, \beta]$ is customized based on how close $\|q_{H^*}\|$ is to q. Also, recall that for a preference top-k query, top-κ objects are computed in each core-subspace, where $\kappa \in [k, \beta k]$ is customized based on how close $\|q_{H^*}\|$ is to q. To test the effectiveness of this customization heuristic, for each covered preference q, again the smallest integer β is computed s.t. Γ_q is β-approximate. Next, those β values are partitioned into bins based on the angle between q and q_{H^*}. More specifically, if the angle between q and q_{H^*} is $\theta°$, q's β is put into bin $\lfloor \theta/5 \rfloor$. For each bin i, the value of β at the 90-th percentile is chosen as an estimated β for every preference whose angle is in within $[5(i-1)°, 5i°)$. Table 4.6.3 shows that for the *box-uniform* workload, the smallest angle for 33.56% of preferences is between $0°$ and $5°$ and at least 90% of those preferences is 1-approximate. Similarly, the smallest angle for 5.18% of preferences is between $5°$ and $10°$ and at least 90% of those preferences is 2-approximate. Table 4.6.3 shows the results for *Sphere-uniform*.

Parameter δ (Algorithm 1) This paragraph shows the results on parameter δ (in Algorithm 1). Here, $d = 80$, $k = 5$, and $10{,}000$ preferences are drawn from 200 skewed

Table 4.1: Box-uniform

Angle (degrees)	$[0, 5)$	$[5, 10)$
Fraction of preferences	0.3356	0.0518
Estimated β	1	2

Table 4.2: Sphere-uniform

Angle (degrees)	$[0, 5)$	$[5, 10)$	$[10, 15)$	$[15, 20)$	$[20, 25)$	$[25, 30)$
Fraction of preferences	0.3678	0.0320	$.0405$	0.0509	0.0516	0.0581
Estimated β	1	1	1	2	2	2

generating subspaces with maximum dimensionality $\tau_{\text{gen}} = 6$. $100{,}000$ objects are drawn from the *box-uniform* distribution. Figures 4.17(a) and 4.17(b) show the number of core subspaces and top-k query time when varying δ, respectively. When $\delta = 0.1$, $|\mathbb{H}| = 47$ and query time is 0.51 milli-seconds. When $\delta = 0.5$, $|\mathbb{H}|$ decreases to 9, but query time increases to 19.96 milli-seconds. This parameter allows users to control trade-offs between space consumption and query time. The reason for the increase in query time is that as δ increases, the fraction of uncovered preferences also increases, as shown in Figure 4.17(c). When $\delta = 0.1$, the 47 core-subspaces cover roughly 93.4% of preferences; but when $\delta = 0.5$, the 9 core-subspaces cover roughly 36% of preferences only. Those uncovered preferences are much slower than the covered preferences because they are handled in the 80-dimensional space. On the other hand, if fewer number of core-subspaces is selected, imperfectly covered preferences will have bigger error. Thus, in Figure 4.17(d), the error deteriorates as δ increases from 0 to 0.1. However, as δ continues to increase, the error becomes significantly better because the top-k queries for uncovered preferences are answered exactly.

Parameter θ (Algorithm 2) This paragraph shows the results on parameter θ (in Algorithm 2). Again, $d = 80$, $k = 5$, and $10{,}000$ preferences are drawn from 200 skewed *generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$. $100{,}000$ objects are drawn

from the *box-uniform* distribution. δ is also fixed to 0.1. Figure 4.18(a) shows that the fraction of covered preferences increases from 0.43 to 1 when θ increases from 0.1 to 0.9. As a result, the average top-k query time drops from 16.26 milli-seconds to 0.36 milli-seconds (Fig 4.18(b)), and the approximation errors deteriorate (Fig 4.18(c) and 4.18(d)) because a preference cover may only cover a small fraction of the preference's weight.

Parameter ν (Algorithm 2) This paragraph shows the results on parameter ν in Algorithm 2. Again, $d = 80$, $k = 5$, $10{,}000$ preferences are drawn from 200 skewed *generating subspaces* with maximum dimensionality $\tau_{\text{gen}} = 6$, and $100{,}000$ objects are drawn from the *box-uniform* distribution. Here, δ and θ are set to 0.1 and 0.5, respectively. Figure 4.19(a) shows that when ν increases from 1 to 10, the fraction of covered preferences also increases from 0.621 to 0.99. In particular, more than 85% of preferences find a cover when ν is set to 3. Note that if ν is large, a preference cover may only cover a small fraction of the preference's weight. Figure 4.19(b)) shows that query time decreases from 12.22 milli-seconds to 0.85 milli-seconds as ν increases. Figures 4.19(c) and 4.19(d) show that the errors deteriorate as ν increases.

4.7 Related Work

Preference top-k and reverse top-k queries. As already discussed in Section 4.1, there has been a lot of work on preference top-k queries [44, 67, 113, 85, 50, 49, 65, 66] and reverse top-k queries [115]. This chapter builds on and compares with the solution presented in the previous chapter, which applied the ideas of coreset and duality transform to the full-dimensional space; this reference also provides additional discussion of and comparison with other previous approaches to top-k and reverse top-k queries.

This chapter has compared the core-subspace solution extensively with the view-based approach [68, 50]. As discussed, in some sense, the core subspaces can be seen as a powerful generalization of views. This chapter also shows how to select such views, a

problem that is not addressed in [50].

The layered-based approaches (e.g., [44]) are essentially the exact counterpart of coresets, and are subsumed by coresets because the latter provides more flexible accuracy/space trade-offs. For this reason, this chapter does not compare directly with the layered-based approach or the hybrid approach [65, 66] that builds on them; their difficulty with high dimensions can be seen from the performance gap between *CSI* and *CSI** in Section 4.6.1.

Top-k queries can be seen as special case of rank aggregation [57], and the Threshold Algorithm [58] is viable option for top-k queries; this chapter compares with *TA* extensively in Section 4.6.1.

Finding interesting subspaces. The task of identifying core subspace is related to the problems of *subspace clustering* (finding all clusters in all subspaces) and *projected clustering* (assigning points to clusters that exist in different subspaces). There has been a lot of work on these problems (see [74] for a survey). In particular, if the subspaces are axis-parallel, the problem is also related to the so-called *row/column-subset selection* problem [51, 62]: given a matrix where rows are objects and columns are features, select a subset of features that are dominant. However, the intended use of the core subspaces warrants the specialized algorithm in Section 4.3, which accounts for the feature of multiple coverage, as well as the fact that the distributions of preferences assigned to a subspace are less of a concern than those of preferences across subspaces.

While core subspaces are chosen to be axis-parallel for reasons of simplicity and robustness against changes in attribute weight distributions, there are some situations for which it may be beneficial to consider subspaces that are arbitrarily oriented. For example, the preference workload may be known and stable. As another example, preferences may not exhibit sparsity in the original space, but do so after some affine transformation. In these situations, the problem of finding arbitrarily oriented subspaces is related to *subspace segmentation*, which seeks to model a set of data points using a union of affine

subspaces (see [15] for a survey). PCA can be seen as a very restrictive special case where all points come from a single affine subspace; as shown in Section 4.6, it is less effective than multiple axis-parallel subspaces. Considering multiple arbitrary core subspaces in the solution remains an interesting problem for future work.

4.8 Conclusion

This chapter proposed a solution, based on the idea of core subspaces, for top-k and reverse top-k queries in high dimensions. The solution presented in this chapter exploits the sparsity in preferences to identify core subspaces, and applies the techniques of coresets and duality transform to index each core subspace as well as the full-dimensional space effectively. As shown by the experimental evaluation, in high dimensions, exact methods are slow, while existing approximation methods suffer from either poor speed (e.g., when using only a single coreset in the full space) or poor accuracy (such as the PCA- and view-based approaches). In contrast, for workloads where preferences are often sparse—a case that arises naturally in practice—the solution presented in this chapter offers a desirable trade-off between speed and accuracy, which makes scalable processing of top-k and reverse top-k queries in high dimensions a reality.

5

Range Top-k Subscriptions

In the previous two chapters, we have discussed how to compute the set of affected subscriptions for data updates. In this chapter, subscriptions are distributed across a wide-area network, so the network aspect also needs to be taken into account. In particular, this chapter considers how to support a large number of range top-k subscriptions for wide-area publish/subscribe. Given an object update, subscriptions need to be notified if their top-k results are changed. Simple solutions include using a content-driven network to notify all subscriptions whose ranges contain the update (ignoring top-k), or using a server to compute only the affected subscriptions and notifying them individually. The former solution generates too much network traffic, while the latter overwhelms the server. This chapter presents a geometric framework for the problem that allows the set of affected subscriptions to be described succinctly with messages that can be efficiently disseminated using content-driven networks. Fast algorithms will be given to reformulate each update into a set of messages whose number is provably optimal, with or without knowing all subscriptions. This chapter also presents extensions to the solution, including an approximate algorithm that trades off between the cost of server-side reformulation and that

of subscription-side post-processing, as well as efficient techniques for batch updates.

5.1 Introduction

Consider a range top-k query over a database of objects (e.g. stocks). The query examines a subset of the objects satisfying a range condition (e.g., stocks with risk rating between medium high and high), and picks the top k objects within this subset by some ranking criterion (e.g., stocks with the k lowest price-to-earning ratios). Over time, when the set of objects or their attribute values change, the query result has to be kept up to date, as in the standard view maintenance and continuous query processing settings. This chapter studies how to support hundreds of thousands or even millions of such queries simultaneously. Representing different user interests, these queries may have different range conditions and therefore different lists of k objects as their answers.

A challenging application setting is when a large number of these queries, which is referred to as *subscriptions*, are located across a wide-area network. For each event updating the database, all subscriptions whose results are affected must be notified. Notification messages should carry enough information so that the affected subscriptions can update their top-k lists accordingly. A naive approach would be to use a central server to maintain all objects and subscriptions, compute the list of affected subscriptions for each event, and notify each affected subscription with the change to its top-k list. Since an event may affect many subscriptions, this approach can easily overload the server with processing and messaging costs at least linear in the number of affected subscriptions.

A solution is to push some event processing and dissemination work into a more "intelligent" network, but at the cost of increasing system complexity. As demonstrated in previous work [41, 40, 42], a *content-driven network* (*CN*) offers a good trade-off between functionality and complexity. CN is a class of overlay networks designed for efficient dissemination, with a clean message interface. Many off-the-shelf overlay networks are ex-

amples of CN, e.g., *content-based networks* [35] and *content-addressable networks* [99].[1] For the purpose of this chapter, CN is regarded to as a black box for efficiently delivering a message to all subscriptions whose query parameters satisfy a selection condition carried by the message.[2] Instead of enumerating affected subscriptions one by one, the server would compute a compact description for the set of affected subscriptions, and then translate this description into a series of condition-carrying messages to be sent through CN. The number of such messages is usually far less than the number of affected subscriptions, thereby relieving the server bottleneck.

Range top-k subscriptions are challenging for several reasons. It is straightforward for CN to handle range subscriptions without top-k as in standard *publish/subscribe*: a message simply needs to list the updated object's attribute values, which can be interpreted as a condition testing whether a subscription range contains the object. However, such a message is not enough for range top-k subscriptions because they are "stateful": whether a subscription is affected depends on how the updated object ranks against others within the subscription range. Furthermore, if the updated object drops out of a subscription's top-k list, the new k-th ranked object must be sent to the subscription. While previous work [41] addresses the special case of $k = 1$ (i.e., range min/max subscriptions), the general case handled in this chapter is considerably more complex and has more practical applications.

A geometric framework. This chapter develops a geometric framework to support range top-k subscriptions. The geometric framework enables the problem of generating notification messages to be viewed intuitively as one of tiling a potentially complex region of affected subscriptions (in an appropriately defined subscription space) using simple geometric shapes. The set of tiles forms a compact description of the region. Each tile cor-

[1] CN is named after these popular examples, which should *not* be confused with *content delivery/distribution networks* [32] that serve the different purpose of replicating popular Web objects.

[2] An equivalent, dual view is that CN allows subscriptions to be selection conditions over message attributes, and CN efficiently delivers a message to all subscriptions whose conditions are satisfied by the message.

responds to a CN message, whose condition selects all subscriptions covered by the tile. While one could first compute the list of affected subscriptions and then find the tiling, this chapter develops algorithms (described below) that avoid computing this potentially long list in the first place.

New algorithms. New algorithms are proposed for message generation based on the framework above. These algorithms are scalable—they run in time dependent on the number of messages they generate, not the number of affected subscriptions (which could be substantially larger). Experiments confirm that this property translates into substantial savings in both server running time and network dissemination cost; furthermore, the performance lead over other approaches widens as the number of subscriptions increases.

This chapter starts with two algorithms. The first one, which is referred to as *Paint-Dense*, is subscription-oblivious; it examines only the set of objects. This feature is attractive from both scalability and privacy perspectives, because it alleviates the need for a server to track a large number of subscriptions. *Paint-Dense* computes the optimal tiling assuming no knowledge of the subscriptions. The second version, *Paint-Sparse*, uses both the set of objects and the set of subscriptions. Intuitively, it produces a tiling sensitive to the subscription distribution; the size of the tiling is 2-approximate and often much smaller than that generated by *Paint-Dense*.

This chapter also considers the case of batch updates, where a subscription needs to be notified of the net change in its result at the end of a batch. Simply processing this batch one event at a time generates more traffic than necessary. It will be seen later that by pre-processing the batch (coalescing and reordering updates), subscribers are guaranteed to receive the minimum number of messages needed.

Besides *Paint-Dense* and *Paint-Sparse*, this chapter provides approximate algorithms that generate even fewer messages from the server at the expense of more "false positives"—notifications received by a subscription but not needed. False positives are discarded by

each subscription with simple local post-processing, so the "approximate" algorithms still guarantee exact subscription results. Having fewer messages reduces processing and messaging loads on the server, but false positives bring higher last-hop traffic and extra post-processing. The trade-off can be adjusted using a parameter $\varepsilon \leq 1$, while guaranteeing that subscriptions miss no notifications and receive no objects ranked below $(1+\varepsilon)k$.

While the focus of this chapter is on 1-d range top-k subscriptions, this chapter also sketches out how our framework and algorithms can be generalized to subscriptions whose range conditions involve multiple dimensions and more general constraints. As a concrete illustration, this chapter also presents the detailed algorithm and experimental evaluation for 1.5-d range top-k subscriptions[3] in the extension section.

The subscription type considered in this chapter—orthogonal range top-k—is a standard one in most subscription/query languages. While there exist a plethora of proposals for other language features, little is known about how best to support this standard subscription type; this chapter will fill this void. Note that the techniques presented in this chapter apply to top-k subscriptions with other types of conditions too. For example, conditions comparing categorical attributes against concepts drawn from a hierarchy can be mapped to range conditions with appropriate encoding of the hierarchy. For another example, range conditions subsume near-neighbor conditions under the L_∞ norm, and in low dimensions they can be effective as building blocks for supporting near-neighbor and nearest-neighbor conditions under other distance metrics.

This chapter focuses on application settings with many geographically dispersed subscriptions to a central database (e.g., news aggregators and financial information services). However, the solution presented in this chapter can be extended to other settings, ranging from simpler ones such as non-distributed continuous query systems with no need to deliver results over a network, to more complex ones such as publish/subscribe systems with

[3] An example of a 1.5-d range top-k subscription would be "k stocks that have the lowest price-to-earning ratio among those with market capitalization above 50 billion US dollars and risk rating between medium high and high."

multiple, distributed event publishers.

5.2 Overview

5.2.1 Problem Formulation

Consider a set $\mathcal{O}$ of n objects. For simplicity, assume each object has only two numeric attributes: x is used in range conditions, while y is used for ranking objects in ascending order of their y-values. Section 5.5 discusses how to generalize the problem and the solutions to higher dimensions. For each object i ($1 \leq i \leq n$), let $x_i \in \mathbb{R}$ denote its x-value and $y_i \in \mathbb{R}$ denote its y-value. Without loss of generality, all x_i's and y_i's are assumed to be distinct.

There is a set $\mathcal{S}$ of m subscriptions over the network. Each subscription S_j ($1 \leq j \leq m$) specifies an x-value range of interest, denoted $\sigma_j = [\ell_j, r_j] \subseteq \mathbb{R}$. For some $k \ll n$, S_j wishes to track the top k objects (along their attribute values) in σ_j, i.e., those with the k smallest y-values. More precisely, S_j must maintain, at all times, the list $\mathrm{top}_k(S_j) = \{(x_i, y_i) \mid x_i \in \sigma_j \wedge |\{i' \mid x_{i'} \in \sigma_j \wedge y_{i'} < y_i\}| < k\}$.

A *(y-update) event*, denoted $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \rightarrow y_i^{\mathrm{new}})$, changes object i's y-value from y_i^{old} to y_i^{new}. Upon receiving an event δ, all *affected* subscriptions must be notified. A subscription S_j is affected by δ iff δ changes $\mathrm{top}_k(S_j)$; i.e., either the membership of this list changes or the y-value of some object in this list is updated as a result of δ. See Figure 5.1(a) for an example. For simplicity of presentation, the discussion will be focused on y-update events.[4]

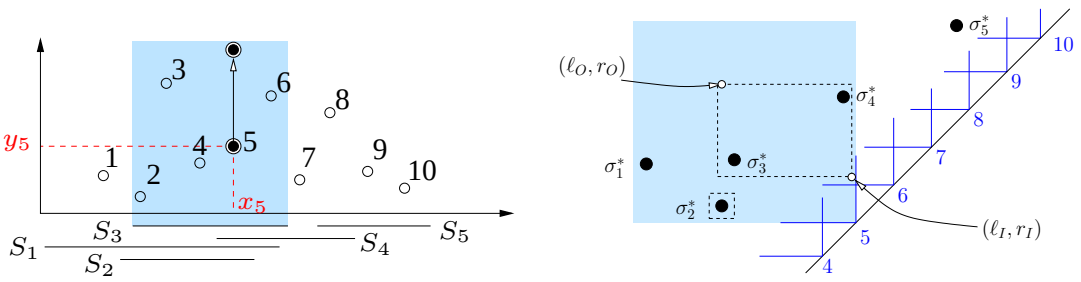
To notify all affected subscriptions, this chapter follows the same overall approach as [41]—first using a server to *reformulate* the event into a sequence of messages, then using CN to *disseminate* these messages to subscriptions, and finally having subscriptions

[4] Object insertion and deletion can be simply treated as y-update events $\mathsf{Upd}(x_i, \infty \rightarrow y_i)$ and $\mathsf{Upd}(x_i, y_i \rightarrow \infty)$, respectively. An update to object i's x-value from x_i^{old} to x_i^{new} can be simulated by a deletion of $(x_i^{\mathrm{old}}, y_i)$ followed by an insertion of $(x_i^{\mathrm{new}}, y_i)$. Alternatively, it is straightforward to extend the algorithms to handle these events directly.

post-process received messages to maintain their top-k lists. More specifically, the server maintains the set of objects $\mathcal{O}$, and reformulates each event into a sequence of constant-size CN messages of the format $\mathsf{Msg}(\ell_I, r_I, \ell_O, r_O, x_i, y_i)$, where $[\ell_I, r_I] \subseteq (\ell_O, r_O)$ are two nested ranges in $\mathbb{R}$, and (x_i, y_i) represents some object i (with its attribute values). Each message is interpreted as a condition over subscriptions' ranges of interest: CN delivers this message to subscription S_j iff $[\ell_I, r_I] \subseteq \sigma_j \subseteq (\ell_O, r_O)$ (see Figure 5.1(b)). Each subscription S_j maintains its own top-k list L_j. Upon receiving a message, S_j checks whether L_j currently contains object i (the one with x-value equal to x_i). If yes, S_j simply updates the y-value of this object to y_i. Otherwise, S_j updates its list L_j to contain the top k objects in $L_j \cup \{(x_i, y_i)\}$.

The goal of this chapter is to develop efficient algorithms for generating the sequence of CN messages for each event, such that every affected subscription will have its top-k list correctly updated by following the protocol above. Several performance measures are considered in designing the algorithms: 1) the number of messages generated; 2) time spent by the server in generating them; and 3) the number of messages received by the subscriptions.[5] These measures present interesting trade-offs and must be considered jointly. For instance, minimizing (3) alone would not be sufficient; the naive approach of enumerating all affected subscriptions and unicasting to them one by one achieves this objective, but does poorly on the other criteria. A better goal is to keep (3) minimized and optimize other criteria as much as possible; the *exact* algorithms presented in this chapter have this goal. If the server becomes a bottleneck, (1) and (2) can be further reduced at the expense of (3). In this case, an unaffected subscription is allowed to be notified; the *approximate* algorithms presented in this chapter take this approach. These results are discussed further below.

[5] For evaluation (Section 5.7), especially comparison with approaches that do not use CN, the total traffic in the underlying IP network is also considered.



(a) Event space $\mathbb{E}$. (b) Subscription space $\mathbb{S}$.

FIGURE 5.1: $\mathcal{O}$ and $\mathcal{S}$ in $\mathbb{E}$ and $\mathbb{S}$. (a) The shaded vertical strip is for subscription S_3. Increasing object 5's y-value as shown would cause 5 to be replaced by 3 in $\text{top}_k(S_2)$, and by 6 in $\text{top}_k(S_3)$ and $\text{top}_k(S_4)$, where $k = 3$. (b) The shaded quadrant is for object 5. A CN message is shown with dashed outline.

5.2.2 *Overview of Algorithms*

Exact algorithms. With an *exact* algorithm, the server generates messages for each event such that only affected subscriptions are notified, and they each receive only one message (per y-update event). Two settings are considered:

- *Subscription-oblivious.* For the case where the server has no knowledge of the set of subscriptions (because of either scalability or privacy concerns), an algorithm *Paint-Dense* is developed with the following properties (Theorems 12 and 13):
  - The algorithm is given $\mathcal{O}$, but not $\mathcal{S}$.
  - It generates the minimum number of messages possible for any exact algorithm if $\mathcal{S}$ is *dense*: that is, given the set of objects $\mathcal{O}$, for any x-value range σ, there exists some subscription interested in precisely the objects within σ.
  - Its running time depends on the number of messages generated, but not on $|\mathcal{S}|$ or the number of affected subscriptions, which can be much larger.
- *Subscription-aware.* A set of subscriptions is called *sparse* if it is not dense. In this case, *Paint-Dense* may generate a message that does not reach any subscription,

wasting both server processing and network dissemination efforts. Therefore, *Paint-Sparse* is developed, with the following properties (Theorems 16 and 17):

- The algorithm is given both $\mathcal{O}$ and $\mathcal{S}$.
- It generates at most twice the minimum number of messages possible for any exact algorithm, and it never generates any message that reaches no subscription.
- Its running time is sublinear in $|\mathcal{O}|$ and $|\mathcal{S}|$, and depends on the number of messages generated instead of the number of affected subscriptions, which can be much larger.

Extensions. This chapter also considers the batched version of the problem, in which subscriptions only need to have their top-k lists correctly updated at the end of an event sequence. *Paint-Batch* is developed to pre-processes the event sequence before applying either algorithm above (with minor modifications) to each event. *Paint-Batch* operates well with the basic CN interface of Section 5.2.1, and is able to guarantee that each subscriber receives the minimum number of messages possible (Theorem 24), which is far less than if all events are processed in the sequence in order.

This chapter also relaxes the requirement that only affected subscription may receive messages. By allowing unaffected subscriptions to receive unnecessary messages, an *approximate* algorithm further reduces the number of messages generated by the server. Approximate algorithms *Paint-Dense*(ε) and *Paint-Sparse*(ε) are developed with parameter $\varepsilon \leq 1$ controlling this trade-off. Compared with their exact counterparts, they reduce the number of messages by a factor of εk while guaranteeing that unnecessarily received objects are ranked within $(1 \pm \varepsilon)k$ (Theorem 25). Furthermore, such objects are automatically ignored by subscriptions following the same protocol in Section 5.2.1, so all results remain accurate at all times.

Both extensions above inherit the efficiency of *Paint-Dense* and *Paint-Sparse*, with running times dependent on the number of messages generated rather than subscriptions affected.

This chapter will also briefly discuss how to extend the problem and framework to higher dimensions, where a subscription's range of interest becomes a d-dimensional region. As a concrete illustration, algorithms will be presented for 1.5-dimensional range subscriptions.

Last but not least, this chapter will also discuss how to extend the problem and framework to the distributed setting with multiple, distributed event publishers.

Data structures. For all algorithms presented in this chapter, the server maintains a data structure indexing the set of objects $\mathcal{O}$ by (x, y) as points in $\mathbb{R}^2$. This index supports the following operations:

- Events that update objects in $\mathcal{O}$.
- $\mathrm{first}_k(x_0, y_0, s)$: Here $(x_0, y_0) \in \mathbb{R}^2$ and $s \in \{\leftarrow, \rightarrow\}$. If s is $\leftarrow$ (resp. $\rightarrow$), then this query finds the first k objects in $\mathcal{O}$ in the southwest (resp. southeast) quadrant with (x_0, y_0) as the apex when proceeding in the $(-x)$-direction (resp. $(+x)$-direction) from (x_0, y_0). If the quadrant contains fewer than k objects, all of them are reported. Only the x-values of the objects are reported by first_k, and they are reported in the order encountered.
- $\min_y(\sigma, y_0)$: Given an x-value range σ and a y-value y_0, this query returns the object in $\mathcal{O}$ with the minimum y-value in the 3-sided rectangle $\sigma \times (y_0, \infty)$.

In this chapter, $t(n)$ (where $n = |\mathcal{O}|$) is used to denote the upper bounds on running times of the operations above: object updates and $\min_y$ all run in $O(t(n))$ time, while first_k runs in $O(t(n) + k)$ time. If kd-tree is used for the index, then the index size is linear and

$t(n) = \sqrt{n}$. If $O(n \log n)$ space is allowed for the index, then a data structure based on *dynamic range trees* [86] can be used to get $t(n) = \log^2 n$.

Algorithms for sparse subscriptions also require a data structure indexing the set of subscriptions $\mathcal{S}$ by (ℓ_j, r_j), the left and right endpoints of their x-value ranges of interest, as points in $\mathbb{R}^2$. This index supports the following operations:

- Insertion and deletion of subscriptions in $\mathcal{S}$.
- $\mathrm{snap}(G)$: Given a rectangle $G \subseteq \mathbb{R}^2$, this query returns the smallest rectangle containing all subscriptions inside R. If there are no such subscriptions, $\emptyset$ is returned.

Using balanced binaries trees, insertion, deletion, and snap can all be processed in $O(\log m)$ time (where $m = |\mathcal{S}|$).

5.3 Geometric Framework

This section introduces a geometric framework essential to the understanding of the problem. Section 5.4 will reveal, with the help of this framework, the structure inherent in the seemingly arbitrary subset of affected subscriptions, which allows the task of generating CN message to be viewed conveniently as one of tiling a complex region using only rectangles.

Event space. Let $\mathbb{E} = \mathbb{R}^2$ denote the *event space*, where each object i is represented as a point $(x_i, y_i) \in \mathbb{R}^2$ (Figure 5.1(a)). Each subscription S_j is interested in objects that lie in the vertical strip $\sigma_j \times \mathbb{R}$; $\mathrm{top}_k(S_j)$ returns the k lowest among them. For an object i and an integer $v > 0$, let $\lambda_v(i)$ (resp. $\varrho_v(i)$) denote the x-coordinate of the v^{th} rightmost (resp. leftmost) object in the southwest (resp. southeast) quadrant with apex (x_i, y_i); if the quadrant contains less than v objects, it is set to $-\infty$ (resp. $+\infty$). The procedure $\mathrm{first}_v(x_i, y_i, \leftarrow)$ returns $\lambda_1(i)$, $\lambda_2(i)$, $\cdots$, $\lambda_v(i)$, and $\mathrm{first}_v(x_i, y_i, \rightarrow)$ returns $\varrho_1(i)$, $\varrho_2(i)$,

$\cdots$, $\varrho_v(i)$. Set $\mathcal{L}(i) = \langle \lambda_v(i) \mid 1 \le i \le k \rangle$ and $\mathcal{R}(i) = \langle \varrho_v(i) \mid 1 \le i \le k \rangle$. The *influence interval* of i, denoted by $\mathrm{II}(i)$, is defined to be

$$\mathrm{II}(i) = [\lambda_k(i), \varrho_k(i)].$$

Lemma 7. *For subscription S_j, if $i \in \mathrm{top}_k(S_j)$, then $\sigma_j \subseteq \mathrm{II}(i)$.*

Proof. If $i \in \mathrm{top}_k(S_j)$, then $x_i \in \sigma_j$ and at most k objects lie in the rectangle $R_j = \sigma \times [-\infty, y_i]$. If the left endpoint of σ_j lies to the left of $\lambda_k(i)$, then R_j contains more than k objects. Similarly, if the left endpoint of σ_j lies to the left of $\varrho_k(i)$, then R_j contains more than k objects. Hence, $\sigma_j \subseteq [\lambda_k(i), \varrho_k(i)]$. □

However, there may be a subscription S_j such that $\sigma_j \subseteq \mathrm{II}(i)$ but $i \notin \mathrm{top}_k(S)$. To fully characterize which subscriptions contain i in their top-k list, we introduce the notion of subscription space and influence region.

Subscription space. Let $\mathbb{S} = \mathbb{R}^2$ denote the *subscription space*, where each subscription with range of interest $\sigma = [\ell, r]$ is mapped to the point $\sigma^* = (\ell, r) \in \mathbb{R}^2$ (Figure 5.1(b)). Object i is mapped to the northwest quadrant θ_i with apex at (x_i, x_i); i.e., $\theta_i = \{(\ell, r) \mid \ell \le x_i \le r\}$. S_j is interested in object i only if $\sigma_j^* \in \theta_i$. A CN message $\mathsf{Msg}(\ell_I, r_I, \ell_O, r_O, x_i, y_i)$ corresponds to notifying, with (x_i, y_i), all subscriptions in the rectangle with southeast and northwest corners at (ℓ_I, r_I) and (ℓ_O, r_O), respectively.

To further capture how the objects' y-values affect their ranking, let $\tilde{\mathbb{S}} = \mathbb{S} \times \mathbb{R}$ denote the *lifted subscription space*, where the third dimension corresponds to y-values and is referred to as the y-axis. For subscription S_j, let $\tilde{\sigma}_j$ be the vertical line passing through σ_j^*; i.e., $\tilde{\sigma}_j = \sigma_j^* \times \mathbb{R}$, oriented in the $(+y)$-direction. For an object i, let $\tilde{\theta}_i$ denote the octant $\theta_i \times [y_i, \infty) = \{(\ell, r, y) \mid \ell \le x_i \le r \wedge y_i \le y\}$, with apex at (x_i, x_i, y_i). A y-value update $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \to y_i^{\mathrm{new}})$ corresponds to translating $\tilde{\theta}_i$ in the y-direction so that its apex moves from $(x_i, x_i, y_i^{\mathrm{old}})$ to $(x_i, x_i, y_i^{\mathrm{new}})$. There is a bijection between the quadrant

θ_i and the bottom face $\theta_i \times \{y_i\}$ of the octant $\tilde{\theta}_i$. If $\tilde{\theta}_i$ is the v^{th} octant intersected by the line $\tilde{\sigma}_i$, going in $(+y)$-direction, then i is the rank-v object among the objects in which S_j is interested. Therefore, $\mathrm{top}_k(S_j)$ is the list of objects corresponding to the first k octants that line $\tilde{\sigma}_j$ intersects in $\tilde{\mathbb{S}}$, going in the $+y$ direction.

The *level* of a point $\xi \in \tilde{\mathbb{S}}$, denoted by $\triangle(\xi)$, is the number of octants in $\{\tilde{\theta}_i \mid 1 \leq i \leq |\mathcal{O}|\}$ that contain ξ. For an object i and an integer $v > 0$, let $\theta_i^v \subseteq \theta_i$ be the set of points in θ_i s.t. the level of the corresponding points on the bottom face of θ_i is v. That is,

$$\theta_i^v = \{\sigma \in \theta_i \mid \triangle((\sigma, y_i)) = v\}.$$

Set $\theta_i^{\leq v} = \bigcup_{0<u\leq v} \theta_i^u$. It can be verified that if $\sigma_j^* \in \theta_i^v$, then i is the rank-v object among the objects in which S_j is interested. Hence if $\sigma_j^* \in \theta_i^{\leq v}$, then $i \in \mathrm{top}_v(S_j)$.

Let $\Theta_i = \{\theta_j \mid y_j < y_i\}$ be the set of quadrants corresponding to the objects whose y-values are smaller than that of i. The quadrants in the set Θ_i partition the quadrant θ_i into a family $\Theta_i^{\square}$ of rectangles such that each rectangle lies in the same subset of Θ_i. For a point $\sigma \in \theta_i$, we define $\triangle_i(\sigma)$ to be 1 plus the number of quadrants of Θ_i that contain σ. Note that $\triangle_i(\sigma)$ is the same for all points in (the interior of) a rectangle $R \in \Theta_i^{\square}$ and we denote this value by $\triangle(R)$. Furthermore, if $\triangle(R) = v$, then $R \subseteq \theta_i^v$. Hence, $\theta_i^{(v)} = \bigcup\{R \in \Theta_i^{\square} \mid \triangle(R) = v\}$. For any point $\sigma \in \theta_i$, as we move in the north or the west direction, the value of $\triangle_i(\sigma)$ cannot increase because if a quadrant contains σ, then it also contains all points that lie in the northwest quadrant with σ as the apex. Hence, the rectangles of $\Theta_i^{\square}$ that lie in θ_i^v form a staircase, and the region $\theta_i^{\leq v}$ is a staircase.

Influence region. For an object i, we define its influence region, denoted by $\mathrm{IR}(i)$, to be $\theta_i^{\leq k}$. The following lemma follows from the above discussion.

Lemma 8. *For any subscription S_j, $i \in \mathrm{top}_k(S_j)$ if and only if $\sigma_j^* \in \mathrm{IR}(i)$.*

In other words, $\mathrm{IR}(i)$ characterizes the set of subscriptions that contain object i in their top-k lists. We next understand the structure of $\mathrm{IR}(i)$.

Let $\mathbb{D}(i) = \{\theta_j \mid x_j \in \mathcal{L}(i) \cup \mathcal{R}(i)\}$. We show that $\mathcal{L}(i)$, $\mathcal{R}(i)$, and $\mathbb{D}(i)$ completely define $\mathrm{IR}(i)$. Let $\mathrm{IR}^{\square}(i)$ be the partition of $\mathrm{IR}(i)$ with rectangles induced by the quadrants of $\mathbb{D}(i)$.

Lemma 9. *(i)* $\mathrm{IR}(i)$ *does not intersect any quadrant of* $\Theta_i \backslash \mathbb{D}(i)$.

(ii) $\mathrm{IR}^{\square}(i) = \mathrm{IR}(i) \cap \Theta_i^{\square}$, *i.e.* $\mathrm{IR}^{\square}(i)$ *is the same as the partition* $\Theta_i^{\square}$ *restricted to* $\mathrm{IR}(i)$.

Proof. (i) Suppose $\mathrm{IR}(i)$ intersect a quadrant $\theta_j \in \Theta_i \backslash \mathbb{D}(i)$. Then $\lambda_k(i) < x_j < \varrho_k(i)$. W.L.O.G., assume $x_j < x_i$. Since $y_j < y_i$ and $\lambda_k(i) < x_j$, object j is one of the $k-1$ rightmost objects in the southwest quadrant with apex (x_i, y_i). Thus, $x_j \in \mathcal{L}(i)$. By definition of $\mathbb{D}(i)$, $\theta_j \in \mathbb{D}(i)$, which is a contradiction.

(ii) Since $\mathrm{IR}(i)$ does not intersect any quadrant of $\Theta_i \backslash \mathbb{D}(i)$, the partition of $\mathrm{IR}(i)$ induced by the quadrants in Θ_i is the same as the partition of $\mathrm{IR}(i)$ induced by the quadrants in $\mathbb{D}(i)$.

□

We next describe the geometric structure of $\mathrm{IR}(i)$.

Lemma 10. *Let* $\ell_1 \geq \ell_2 \geq \cdots \geq \ell_k$ *be the values in* $\mathcal{L}(i)$ *and* $r_1 \leq r_2 \leq \cdots \leq r_k$ *be the values in* $\mathcal{R}(i)$. *Set* $\ell_0 = r_0 = x_i$. *Then* $\mathrm{IR}(i)$ *is a staircase polygon with vertices* $(\ell_k, r_0), (\ell_k, r_1), (\ell_{k-1}, r_1), \ldots, (\ell_1, r_{k-1}), (\ell_0, r_k), (\ell_0, r_k)$.

Proof. For $0 < u, v < k$, a point in the rectangle $[\ell_{u+1}.\ell_u) \times [r_v, r_{v+1})$ lies in u quadrants of $\mathbb{D}(i)$ that lie above θ_i, so for $0 < u \leq k$, $\triangle_i(\xi) = k$ for all points $\xi \in [\ell_u, \ell_{u-1}) \times [r_{k-u}, r_{k-u+1})$. Hence, (ℓ_u, r_{k-u}), (ℓ_u, r_{k-u+1}), and (ℓ_{u-1}, r_{k-u+1}) are vertices of $\mathrm{IR}(i)$.

□

An example of $\mathrm{IR}(i)$ is shown in Figure 5.2, in which $k = 5$ and the numbers indicate the level number of rectangles of $\Theta_i^{\square}$. Given $\mathcal{L}(i)$ and $\mathcal{R}(i)$, $\mathrm{IR}(i)$ and $\mathrm{IR}^{\square}(i)$ can be computed in $O(k)$ and $O(k^2)$ time, respectively. Finally, we describe how $\mathrm{IR}(i)$ changes

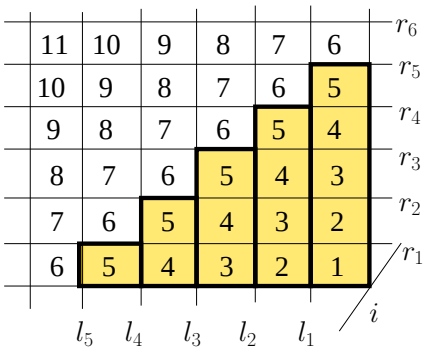


FIGURE 5.2: Tiling $\mathrm{IR}^{\mathrm{new}}(i)$ (shaded, $k = 5$) by CN messages (shown with thick outlines).

as we increase (or decrease) the value of y_i. Suppose we change y_i from y_i^{old} to y_i^{new}. If the set $\mathbb{D}(i)$ does not change as we vary y_i from y_i^{old} to y_i^{new}, $\mathrm{IR}(i)$ does not change (by Lemmas 9 and 10), and thus the top-k list does not change for any subscription (by Lemma 8). The set $\mathbb{D}(i)$ changes when $\mathcal{L}(i)$ or $\mathcal{R}(i)$ changes, which happens when y_i becomes equal to y_j for some object j such that $x_j \in \mathrm{II}(i)$. We say that the object i *encounters* object j when this happens. Object j is referred to as an *exposed* object. We describe how $\mathrm{IR}(i)$ and $\mathrm{IR}(j)$ change when i encounters j, i.e., when y_i changes from $y_i^{\mathrm{old}} = y_j - \epsilon$ to $y_i^{\mathrm{new}} = y_j + \epsilon$ for some sufficiently small ϵ such that no other objects has its y-value in the interval $[y_i^{\mathrm{old}}, y_i^{\mathrm{new}}]$.

For an object s, let $K_s = \theta_s^k$. We define $\mathbb{D}^{\mathrm{old}}(i)$, $\mathbb{R}^{\mathrm{old}}(i)$, K_i^{old} (resp. $\mathbb{D}^{\mathrm{new}}(i)$, $\mathbb{R}^{\mathrm{new}}(i)$, K_i^{new}) to be $\mathbb{D}(i)$, $\mathbb{R}(i)$, and K_i for $y_i = y_i^{\mathrm{old}}$ (resp. $y_i = y_i^{\mathrm{new}}$). Similarly, we define these sets for object j. Then $\mathbb{D}_i^{\mathrm{new}} = \mathbb{D}_i^{\mathrm{old}} \cup \{j\}$ and $\mathbb{D}_j^{\mathrm{new}} = \mathbb{D}_j^{\mathrm{old}} \backslash \{i\}$.

Lemma 11. *Let* $\mathbb{K} = K_i^{\mathrm{old}} \cap \theta_j$. *Then* $\mathrm{IR}(i) = \mathrm{IR}^{\mathrm{old}}(i) \backslash \mathbb{K}$ *and* $\mathrm{IR}^{\mathrm{new}}(j) = \mathrm{IR}^{\mathrm{old}}(j) \cup \mathbb{K}$.

Proof. For any point $\xi \in \mathrm{IR}^{\mathrm{old}}(i)$, $\triangle_i(\xi)$ remains the same for $y_i = y_i^{\mathrm{old}}$ and $y_i = y_i^{\mathrm{new}}$ if $\xi \notin \theta_j$ but it increases by 1 if $\xi \in \theta_j$. Hence, $\triangle_i(\xi)$ becomes $k+1$ for all $\xi \in \mathbb{K}$ and thus $\xi \notin \mathrm{IR}^{\mathrm{new}}(i)$. This proves that $\mathrm{IR}^{\mathrm{new}}(i) = \mathrm{IR}^{\mathrm{old}}(i) \backslash \mathbb{K}$.

On the other hand, for any point $\eta \in \theta_j$, $\triangle_j(\eta)$ remains the same if $\eta \notin \theta_i$ but decreases by 1 if $\xi \notin \theta_i$ (since $\theta_i \notin \mathbb{D}^{\mathrm{new}}(j)$). Since the y-value of no other object lies in the range

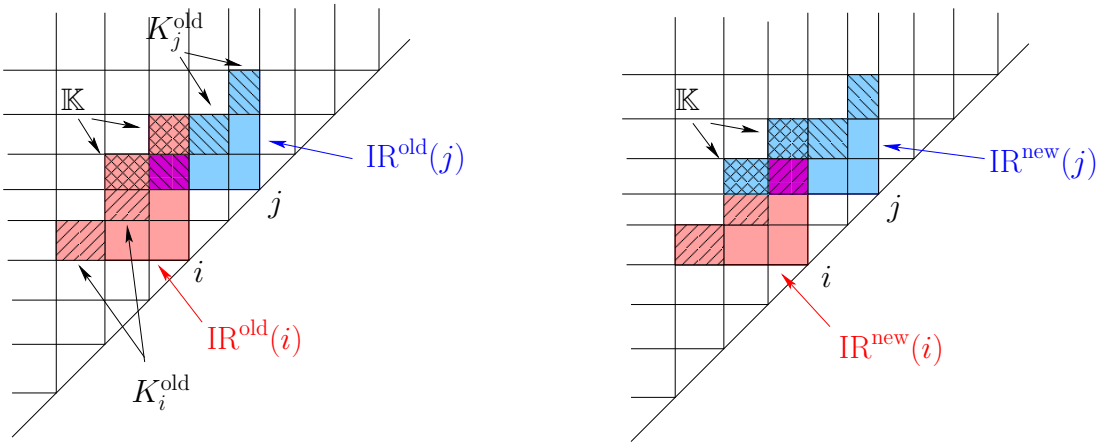


FIGURE 5.3: $\text{IR}(i)$, $\text{IR}(j)$, and $\mathbb{K}$ are shown as the pink, blue, and double dashed regions, respectively. Object i (resp. j) ranks k-th w.r.t. all preferences in the dashed pink (resp. blue) region. When object i encounters object j, $\text{IR}(i)$ shrinks and $\text{IR}(j)$ expands as shown in the right figure.

$[y_i^{\text{old}}, y_i^{\text{new}}]$, for a point $\eta \in \theta_j \cap \theta_i$, if a quadrant $\theta_s \in \mathbb{D}^{\text{old}}(j)$ for $s \neq i, j$, contains η then $\theta_s \in \mathbb{D}^{\text{old}}(i)$ and vice-versa. Hence, $\triangle_j(\eta) = \triangle_i^{\text{old}}(\eta) + 1$ and $\triangle_j(\eta) = \triangle_j^{\text{old}}(\eta) - 1 = \triangle_i^{\text{old}}(\eta) = \triangle_i^{\text{new}}(\eta) - 1$. In other words, if $\eta \in \mathbb{K}$, i.e. $\triangle_i^{\text{old}}(\eta) = k$, then$\triangle_j^{\text{new}}(\eta) = k$ and $\eta \in \text{IR}^{\text{new}}(j)$. Consequently, $\text{IR}^{\text{new}}(j) = \text{IR}^{\text{old}}(j) \cup \mathbb{K}$. □

Figure 5.3 demonstrates how $\text{IR}(i)$ and $\text{IR}(j)$ change as the value of y_i increases. By Lemma 11, $\text{IR}^{\text{new}}(i) \oplus \text{IR}^{\text{old}}(i) = \text{IR}^{\text{new}}(j) \oplus \text{IR}^{\text{old}}(j) = \mathbb{K}$. We note that given $\mathcal{L}(i)$,$\mathcal{R}(i)$, and K_i, $\mathbb{K}$ can be computed in $O(k)$ time.

Finally, we note that the change in the influence region as object i encounters object j while we decrease the y-value of i is similar—switch the role of i and j. The influence intervals of i and j also change. Suppose $x_j > x_i$. Then $\lambda_k^{\text{old}}(i) = \lambda_k^{\text{new}}(i)$ but $\varrho_k^{\text{new}}(i) = \max\{x_j, \lambda_{k-1}^{\text{old}}(i)\} < \varrho_k^{\text{old}}(i)$, and $\varrho_k^{\text{new}}(j) = \varrho_k^{\text{old}}(j)$ and $\lambda_k^{\text{new}}(i) = \lambda_{k+1}^{\text{old}}(i)$.

Example. Refer to Fig. 5.1 and 5.4. All subscriptions other than S_5 are interested in object 5. In $\mathbb{S}$, the northwest quadrant θ_5 contains σ_1^*, σ_2^*, σ_3^*, σ_4^*, but not σ_5^*. Suppose $k = 3$. The influence region of object 5 is an axis-aligned subregion of the quadrant with vertices (x_5, x_5), (ℓ_3, x_5), (ℓ_3, r_1), (ℓ_2, r_1), (ℓ_2, r_2), (ℓ_1, r_2), (ℓ_1, r_3), (x_5, r_3) in clockwise order, where ℓ_1, ℓ_2, and ℓ_3 (resp. r_1, r_2, r_3) are the

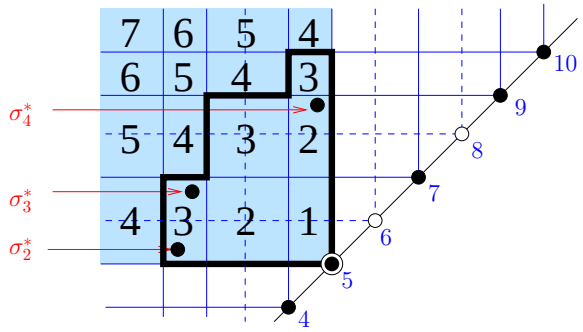


FIGURE 5.4: Partitioning of quadrant θ_5 (shaded) in $\mathbb{S}$ for object 5. Rectangles are shown with level numbers. Objects that do not contribute to this partitioning (because they have larger y-values than that of 5) are shown as circles and with dashed-line quadrants. The influence region of 5, $\mathrm{IR}(5)$, for $k = 3$, is shown with thick outline.

x-values of objects 4, 2, and 1 (resp. objects 7, 9, and 10), respectively. Since S_2 and S_3 rank object 5 at third, σ_2^* and σ_3^* lie in $\theta_5^3 \subset \mathrm{IR}(5)$. Similarly, S_4 ranks object 5 at second and σ_4^* lies in $\theta_5^2 \subset \mathrm{IR}(5)$. However, $\sigma_1^* \in \theta_5^4 \not\subset \mathrm{IR}(5)$ because S_1 ranks object 5 at fourth and therefore, object 5 is not in $\mathrm{top}_k(S_1)$. When object 5's y-value increases as shown, object 5 is replaced by object 3 in $\mathrm{top}_k(S_2)$, and by object 6 in $\mathrm{top}_k(S_3)$ and $\mathrm{top}_k(S_4)$. CN messages (dashed rectangles) are generated to notify S_2, S_3 and S_4.

5.4 Exact Algorithms

5.4.1 Subscription-Oblivious

Consider an event $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \rightarrow y_i^{\mathrm{new}})$, which moves the octant $\tilde{\theta}_i$ in the vertical direction from position y_i^{old} to y_i^{new}. Let $\mathrm{IR}^{\mathrm{old}}(i)$ (resp. $\mathrm{IR}^{\mathrm{new}}(i)$) denote the influence region of object i before (resp. after) the update. There are two cases: $y_i^{\mathrm{old}} > y_i^{\mathrm{new}}$, which possibly raises object i's rank, and $y_i^{\mathrm{old}} < y_i^{\mathrm{new}}$, which possibly lowers object i's rank.

Rank-raising update. This case is simple. It can be easily seen that if $y_i^{\mathrm{old}} > y_i^{\mathrm{new}}$, then $\mathrm{IR}^{\mathrm{new}}(i) \supseteq \mathrm{IR}^{\mathrm{old}}(i)$. Every subscription S_j in $\mathrm{IR}^{\mathrm{old}}(i)$ (i.e., $\sigma_j^* \in \mathrm{IR}^{\mathrm{old}}(i)$) must receive $(x_i, y_i^{\mathrm{new}})$ to update the y-value of object i in $\mathrm{top}_k(S_j)$. Every subscription S_j in $\mathrm{IR}^{\mathrm{new}}(i) \setminus \mathrm{IR}^{\mathrm{old}}(i)$ must receive $(x_i, y_i^{\mathrm{new}})$ as a new object in $\mathrm{top}_k(S_j)$, which would displace some other object from $\mathrm{top}_k(S_j)$. In sum, it suffices to notify all subscriptions in $\mathrm{IR}^{\mathrm{new}}(i)$ with $(x_i, y_i^{\mathrm{new}})$. Since each CN

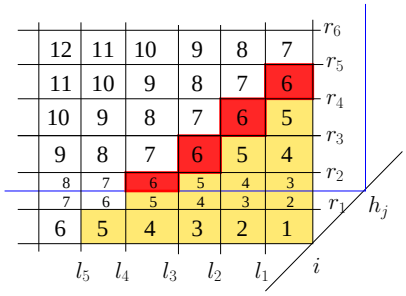


FIGURE 5.5: Effect on $\mathrm{IR}^z(i)$ of encountering exposed object h_j during the sweep. Before the encounter, $\mathrm{IR}^z(i)$ contains both darkly and lightly shaded rectangles (level numbers before the encounter are shown in Figure 5.2); after the encounter, $\mathrm{IR}^z(i)$ contains only the lightly shaded rectangles. The difference, which is gained by h_j as $\mathrm{IR}^{\mathrm{new}}(h_j) \setminus \mathrm{IR}^{\mathrm{old}}(h_j)$, is tiled by CN messages shown with thick outlines.

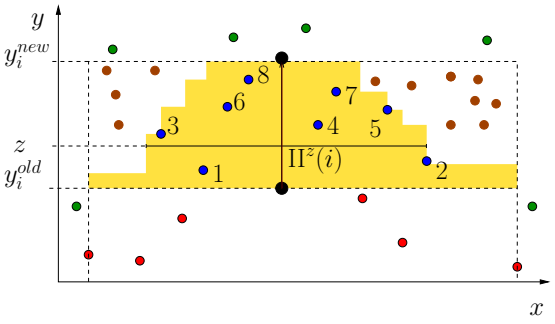


FIGURE 5.6: Sweep in $\mathbb{E}$ ($k = 3$). The width of the shaded area at $y = z$ corresponds to $\mathrm{II}^z(i)$. Exposed objects are numbered in the order encountered.

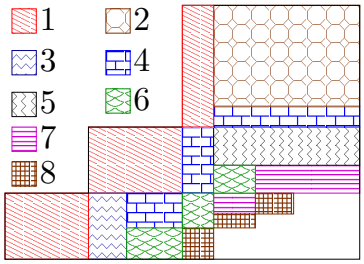


FIGURE 5.7: Tiling $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ by CN messages. Rectangles with the same fill pattern are for the same exposed object.

message reaches a rectangle in $\mathbb{S}$, and $\mathrm{IR}^{\mathrm{new}}(i)$ has up to k "steps," in the worst case k messages are needed to tile $\mathrm{IR}^{\mathrm{new}}(i)$, as illustrated in Figure 5.2. The detailed algorithm, *Paint-Dense-IR*, is presented in Algorithm 3.Its running time, dominated by the two first_k calls to compute the new $\mathrm{IR}(i)$, is $O(t(n) + k)$.

Rank-lowering update. This case is more complex. If $y_i^{\mathrm{old}} < y_i^{\mathrm{new}}$, then $\mathrm{IR}^{\mathrm{new}}(i) \subseteq \mathrm{IR}^{\mathrm{old}}(i)$. First, all subscriptions in $\mathrm{IR}^{\mathrm{old}}(i)$ are notified with $(x_i, y_i^{\mathrm{new}})$ using no more than k messages, in the same way as $\mathrm{IR}^{\mathrm{new}}(i)$ is tiled for a rank-raising update. These messages allow subscriptions to update the y-value of object i in their top-k lists. For those in $\mathrm{IR}^{\mathrm{new}}(i)$, no more messages are needed.

Next, for each subscription in $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$, it needs to further receive an object that will

Algorithm 3: *Paint-Dense*(x_i, y_i^{old}, y_i^{new})

```
1  begin
2      if Rank-Raising Update then
3          L ← first_k(x_i, y_i^new, ←); R ← first_k(x_i, y_i^new, →) ;
4          Paint-Dense-IR(i, L, R) ;
5      else if Rank-Lowering Update then
6          L ← first_k(x_i, y_i^old, ←); R ← first_k(x_i, y_i^old, →) ;
7          Paint-Dense-IR(i, L, R) ;
8          II = conv(L ∪ R); v ← y_i^old ;
9          while v < y_i^new do
10             h_j ← min_y(II, v) ;
11             Paint-Dense-Exposed(h_j, i, L, R) ;
12             if x_{h_j} < x_i then
13                 L ← L ∪ {x_{h_j}} ;
14                 if |L| > k then
15                     deleteLast(L) ;
16             else
17                 R ← R ∪ {x_{h_j}};
18                 if |R| > k then
19                     deleteLast(R);
20             II = conv(L ∪ R); v ← y_{h_j} ;
21 end
```

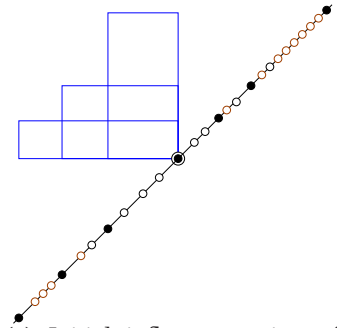
replace i in its top-k list.[6] As described in the previous section, such objects are exposed by the ranking-lowering update, and clearly must have their influence regions expanded. The task then is to notify the subscriptions in $\text{IR}^{\text{old}}(i) \setminus \text{IR}^{\text{new}}(i)$ with respective exposed objects.

Imagine that object i's y-value increases continuously from y_i^{old} to y_i^{new}, i.e., sweeping the octant $\tilde{\theta}_i$ from its old position to its new position in $\tilde{\mathbb{S}}$. Let $\text{IR}^z(i)$ and $\text{II}^z(i)$ denote the influence region and influence interval of i when its y-value is set to z. By Lemma 11, areas gradually "lost" during the sweep by $\text{IR}^z(i)$ (which starts out as $\text{IR}^{\text{old}}(i)$ and eventually shrinks to $\text{IR}^{\text{new}}(i)$) are "gained" by exposed objects' influence regions, as shown in Figure 5.5. For each exposed object h_j, consider the point $z = y_{h_j} - \epsilon$ right before crossing. Any subscription S in $\theta_i^k \cap \theta_{h_j} \subseteq \text{IR}^z(i) \cap \theta_{h_j}$ is interested in both objects i and h_j, and i ranks the k-th in $\text{top}_k(S)$. When z changes from $y_{h_j} - \epsilon$ to $y_{h_j} + \epsilon$, objects i and h_j swap their ranks, and h_j would enter $\text{top}_k(S)$ as the result of the update. On the other hand, for any unexposed object h, when z crosses y_h during the sweep (if

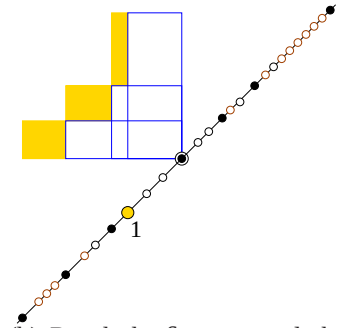
[6] Note that this subscription must receive (x_i, y_i^{new}) before receiving the replacement object; otherwise, the replacement object would appear to be out of the top-k list because of the stale y-value of i.

at all), $\mathrm{IR}^z(i) \cap \theta_h = \emptyset$, implying that object i ranks strictly lower than the k-th for subscriptions interested in both i and h; therefore, swapping i and h's ranks would not put h into any top-k list.

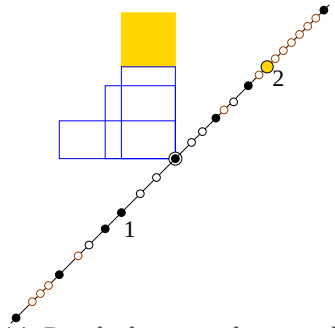
The algorithm is now described in more detail. During the sweep, the algorithm maintains the list $\mathcal{L}^z$ (resp. $\mathcal{R}^z$), which is initialized by $\mathrm{first}_k(x_i, y_i^{\mathrm{old}}, \leftarrow)$ (resp. $\mathrm{first}_k(x_i, y_i^{\mathrm{old}}, \rightarrow)$) and always contains the x-values of the first k objects in $\mathcal{O}$ to the west (resp. east) of x_i with y-values less than z, padded with $-\infty$ (resp. ∞) if there are fewer than k such objects. By Lemma 10, $\mathcal{L}^z$ and



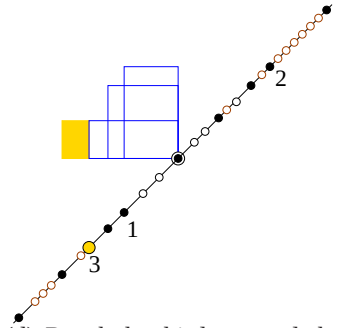
(a) Initial influence region of object i, $\mathrm{IR}(i)$.



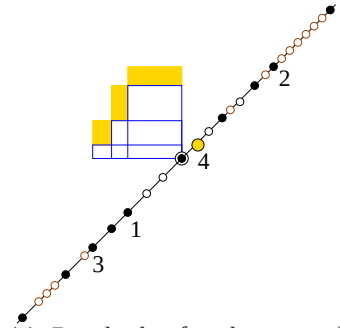
(b) Reach the first exposed object.



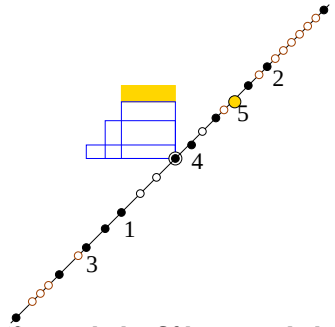
(c) Reach the second exposed object.



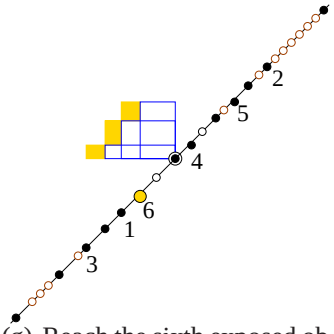
(d) Reach the third exposed object.



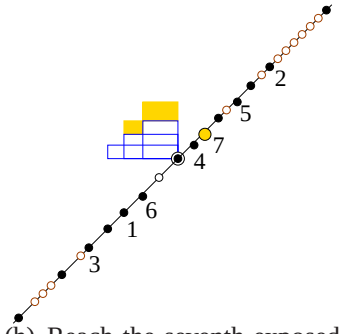
(e) Reach the fourth exposed object.



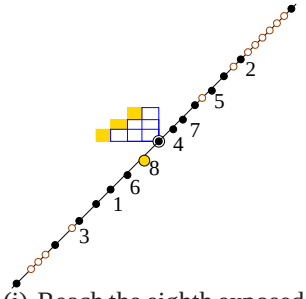
(f) Reach the fifth exposed object.



(g) Reach the sixth exposed object.



(h) Reach the seventh exposed object.



(i) Reach the eighth exposed object.

FIGURE 5.8: Illustration of the rank-lowering update shown in Figure 5.6.

$\mathcal{R}^z$ allow us to readily obtain $\mathrm{IR}^z(i)$, $\mathrm{II}^z(i)$, and the partitioning $\mathrm{IR}^z_\square(i)$ of $\mathrm{IR}^z(i)$ as needed. The next exposed object above z corresponds to the object with the minimum y-value in the 3-sided rectangle $\mathrm{II}^z \times (z, \infty)$ in $\mathbb{E}$, and can be found by $\min_y(\mathrm{II}^z(i), z)$, as illustrated in Figure 5.6.

Say the exposed object found is h_j. $\mathcal{L}^z$ and $\mathcal{R}^z$ are incrementally updated by adding x_{h_j} to the appropriate list ($\mathcal{L}^z$ if $x_{h_j} < x_i$, or $\mathcal{R}^z$ otherwise), and removing from that list the x-value furthest from x_i. Lemma 10 tells us how this incremental update to $\mathcal{L}^z$ and $\mathcal{R}^z$ shrinks $\mathrm{IR}^z(i)$. The area lost from $\mathrm{IR}^z(i)$ is shaped as a series of up to k rectangles along a diagonal in the northeast direction, as illustrated in Figure 5.5. Specifically, the algorithm "paints" over the intersection of $\mathrm{IR}^z(i)$ and θ_{h_j} (quadrant of the exposed object), incrementing the level numbers by 1. Rectangles in the updated $\mathrm{IR}^z_\square(i)$ with level greater than k should be removed from $\mathrm{IR}^z(i)$. By Lemma 11, these rectangles together form $\mathrm{IR}^{\mathrm{new}}(h_j) \setminus \mathrm{IR}^{\mathrm{old}}(h_j)$. Hence, one CN message is generated for each such rectangle with the exposed object values (x_{h_j}, y_{h_j}). Figures 5.8 illustrate the sweep procedure in $\mathbb{S}$.

Algorithm 4: *Paint-Dense-IR*(i, $\mathcal{L}$, $\mathcal{R}$)

```
1  begin
2      M ← {} // Rectangles that only contain affected subscriptions ;
3      a ← |L|; b ← |R|;
4      if a + b < k then
5          M ← M ∪ {Msg(x_i, x_i, −∞, ∞, x_i, y_i)} ;
6      else if a = 0 then
7          M ← M ∪ {Msg(x_i, x_i, −∞, r_k, x_i, y_i)} ;
8      else
9          if b < k then
10             M ← M ∪ {Msg(x_i, x_i, ℓ_{k−b}, ∞, x_i, y_i)} ;
11         if a < k then
12             M ← M ∪ {Msg(ℓ_a, x_i, −∞, r_{k−a}, x_i, y_i)} ;
13         z ← k + 1 − b;
14         while z ≤ a do
15             M ← M ∪ {Msg(ℓ_{z−1}, x_i, ℓ_z, r_{k+1−z}, x_i, y_i)} ;
16             z ← z + 1 ;
17     GenerateMsg(M) // Generate messages ;
18 end
```

When the sweep stops at $z = y_i^{\mathrm{new}}$, $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ will have been completely tiled by messages associated with exposed objects, as shown in Figure 5.7. The complete algorithm, *Paint-Dense*, is presented in Algorithm 3.Processing each exposed object h_j takes $O(t(n) + \mu_j)$ time

Algorithm 5: *Paint-Dense-Exposed*(h_j, i, $\mathcal{L}$, $\mathcal{R}$);

```
begin
    M ← {} // Rectangles that only contain affected subscriptions ;
    a ← |L|; b ← |R|;
    if x_{h_j} > x_i then
        if a + b < k then
            M ← M ∪ {Msg([x_i, x_{h_j}], [−∞, ∞], (x_{h_j}, y_{h_j}))} ;
        else if a = 0 then
            M ← M ∪ {Msg([x_i, x_{h_j}], [−∞, r_k], (x_{h_j}, y_{h_j}))} ;
        else
            if b < k then
                M ← M ∪ {Msg([x_i, x_{h_j}], [ℓ_{k−b}, ∞], (x_{h_j}, y_{h_j}))} ;
            if a < k then
                M ← M ∪ {Msg([ℓ_a, x_{h_j}], [−∞, r_{k−a}], (x_{h_j}, y_{h_j}))} ;
            z ← k + 1 − b;
            while z ≤ a do
                M ← M ∪ {Msg([ℓ_{z−1}, r_{k−z}], [ℓ_z, r_{k+1−z}], (x_{h_j}, y_{h_j}))} ;
                z ← z + 1 ;
    else
        // The "x_{h_j} < x_i" case is symmetric to the "x_{h_j} > x_i" case. ;
    GenerateMsg(M) // Generate messages ;
end
```

where $\mu_j \leq k$ is the number of messages generated for h_j. Therefore, tiling $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ takes $O(\nu t(n) + \sum_{1 \leq j \leq \nu}(\mu_j + \log k))$ time, where ν is the number of exposed objects. Initializing $\mathcal{L}^z$ and $\mathcal{R}^z$ for the sweep and tiling $\mathrm{IR}^{\mathrm{old}}(i)$ take $O(t(n)+k)$ time, so the overall time is $O(\nu(t(n)+\log k) + \mu + k)$, where μ is the total number of messages generated.

Discussion. *Paint-Dense*'s time complexity is summarized below.

Theorem 12. Paint-Dense *runs in time $O(t(n)+k)$ for a rank-raising update, and $O(\nu t(n)+\mu+k)$ time for a rank-lowering update, where μ is the number of messages generated and ν is the number of objects exposed by a rank-lowering update. For a rank-lowering update, $\nu < \mu \leq (\nu+1)k$.*

Proof. As shown in Algorithm 3, a rank-raising update involves two first_k calls, each of which takes $O(t(n)+k)$ time, and one *Paint-Dense-IR* call which takes $O(\mu)$ time, where $\mu \leq k$. Thus, the running time of a rank-raising update is $O(t(n)+k)$. For a rank-lowering update, initializing $\mathcal{L}^z$ and $\mathcal{R}^z$ for the sweep and tiling $\mathrm{IR}^{\mathrm{old}}(i)$ take $O(t(n)+k)$ time. Processing each exposed

object h_j takes $O(t(n) + \mu_j)$ time, where $\mu_j \leq k$ is the number of messages generated for h_j. The $O(t(n))$ term comes from one $\min_y$ call. The insertion or deletion of an element from $\mathcal{L}^z$ and $\mathcal{R}^z$ takes $O(\log k)$ time, which is dominated by $t(n)$. Thus, tiling $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ takes $O(\nu t(n) + \sum_{1 \leq j \leq \nu} \mu_j)$ time, where ν is the number of exposed objects. The overall time is $O(\nu t(n) + \mu + k)$, where μ is the total number of messages generated. In addition, $\nu < \mu \leq (\nu + 1)k$. The first inequality follows from the fact that at least one message is generated for each exposed object and object i itself. The second inequality follows from the fact that at most k messages is generated for object i and each exposed object. □

If subscriptions are not allowed to receive false positives, *Paint-Dense* is optimal in the number of messages that it generates for dense subscriptions.

Theorem 13. *For dense subscriptions, the number of CN messages generated by* Paint-Dense *is the minimum possible for any exact algorithm.*

Proof. It is trivial to show that both Algorithms 4 and 5 generate the minimum number of messages for any object. The theorem immediately follows from the fact that messages are only generated for object i and the exposed objects. □

The next result reveals the inherent complexity in handling range top-k subscriptions. Although the worst case for a rank-lowering update event can be quite bad (exposing $\Theta(|\mathcal{O}|)$ objects), it is not expected to be common in practice, as stated by the following lemma:

Lemma 14. *$O(k)$ objects are injected into network if the object whose value is increased is chosen uniformly at random.*

Proof. For an object i, let η_i be the number of objects j such that increasing the value of j to ∞ causes Algorithm 3 to inject a message involving the object i. Then the expected number of objects injected by the rank-lowering update is bounded by $\sum_{i=1}^{n} \eta_i / n$. Moreover, if increasing the value of j injects a message involving i, then $y_j < y_i$ and the event expands the influence region $\mathrm{IR}(i)$. This happens only when $x_j \in \mathcal{L}(i) \cup \mathcal{R}(i)$ before the event but not after its y-value has increased. Since $|\mathcal{L}(i) \cup \mathcal{R}(i)| \leq 2k$, $\eta_i \leq 2k$, and thus the expected number of objects injected is $O(k)$. □

In fact, as the following theorem shows, the expected number of messages is only $\Theta(k^2)$ if objects to be updated are picked randomly.

Theorem 15. *For any exact algorithm given dense subscriptions, a rank-raising update event requires $\Theta(k)$ CN messages and a rank-lowering update event requires $\Theta(nk)$ CN messages in the worst case. If each rank-lowering update event chooses an object to update uniformly at random, the expected number of CN messages required is $\Theta(k^2)$.*

Proof. Recall that Algorithms 4 and 5 generate at most k messages for object i and each exposed object. Decreasing the y-value of object i injects only one object, namely i itself. In the worst case, increasing the value of an objects causes all n objects to be exposed. By Lemma 14, the expected number of objects injections is $O(k)$. This completes the proof of the theorem.

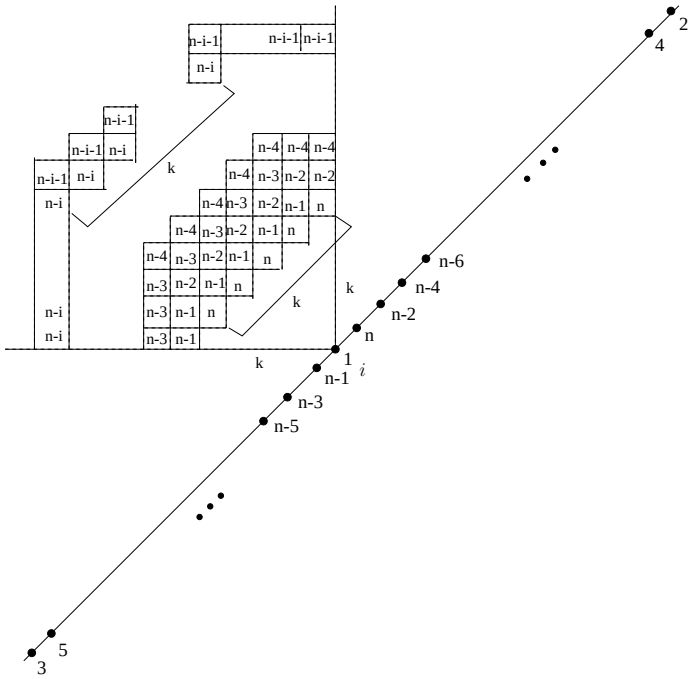


FIGURE 5.9: Lower bound construction: y-values are written along the diagonal line; number inside each rectangle is the level of object i.

Finally, Figure 5.9 shows that if the value of i is increased from 1 to $n+2$, $\Omega(nk)$ messages need to be injected into the network, namely, one for each rectangle in the figure. The same example also shows the bound on the expected number of messages is also tight. □

5.4.2 Subscription-Aware

If the server is given the knowledge about the distribution of subscriptions, the number of CN messages generated by the server can be reduced. In particular, the algorithm can avoid sending

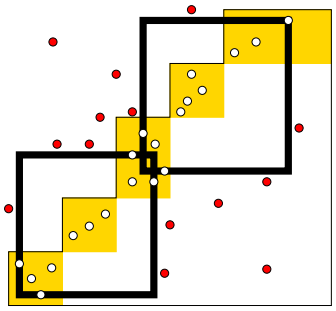


FIGURE 5.10: Reducing the number of rectangles covering $\mathcal{P}$. Subscriptions in $\mathcal{P}$ are shown as circles while those in $\mathcal{S} \setminus \mathcal{P}$ are shown as dots. Rectangles in $\mathcal{G}^{\text{dense}}$ are shaded; rectangles in the optimal covering are shown with thick outlines.

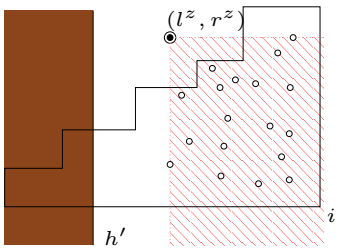


FIGURE 5.11: Finding the next interesting exposed object. The staircase is the current $\text{IR}^z(i)$. Circles represent subscriptions inside $\text{IR}^z(i)$, and are enclosed by the dashed quadrant with apex at (ℓ^z, r^z). Object h', with the darkly shaded quadrant, is an example of an exposed, but inessential object.

messages whose corresponding rectangles in $\mathbb{S}$ contain no subscriptions, and can combine multiple messages into one as long as their bounding rectangles contain no extraneous subscriptions and they carry the same object values.

The general problem can be formulated as a geometric optimization problem: *Given a subset of subscriptions $\mathcal{P} \subseteq \mathcal{S}$ (to notify with the same object values), find a set of rectangles $\mathcal{G}$ in $\mathbb{S}$ such that every point of $\mathcal{P}$ lies in exactly one rectangle of $\mathcal{G}$ and no point of $\mathcal{S} \setminus \mathcal{P}$ lies in any rectangle of $\mathcal{G}$. The goal is to minimize the number of rectangles in $\mathcal{G}$.* Figure 5.10 illustrates this problem. A brute-force approach is to compute the set $\mathcal{P}$ and then solve the standard rectangular covering problem on $\mathcal{P}$. However, doing so requires us to enumerate potentially large sets of affected subscriptions, which we would like to avoid, and this problem is NP-complete in general [12].

A better approach would be to take a list of (at most k) rectangles $\mathcal{G}^{\text{dense}}$ produced by *Paint-Dense* (corresponding to a list of messages with the same object values) as a compact description of $\mathcal{P} = \mathcal{G}^{\text{dense}} \cap \mathcal{S}$, and then solve the problem on $\mathcal{G}^{\text{dense}}$ with the knowledge of $\mathcal{S}$. A simple solution is to go through each rectangle $G \in \mathcal{G}^{\text{dense}}$ and set G to $\text{snap}(G)$ on $\mathcal{S}$; if $\text{snap}(G) = \emptyset$ (i.e., G contains no subscriptions), simply discard G. However, this solution misses the opportunity to combine multiple rectangles into one without introducing false positives, as illustrated in Figure 5.10. Furthermore, it is possible that the entire $\mathcal{G}^{\text{dense}}$ produced by *Paint-Dense* for an exposed object contains no subscriptions, in which case we would like to avoid examining this exposed object and generating $\mathcal{G}^{\text{dense}}$ in the first place.

Algorithm *Paint-Sparse* achieves both goals above. To achieve the first goal of being able to merge rectangles, a greedy approach is taken. Recall from Section 5.4.1 that *Paint-Dense* generates either a list of south-north rectangles forming staircase (for an updated object's influence region) or a list of rectangle forming a diagonal chain (for an exposed object's gain in influence region). In either case, the given rectangles $\mathcal{G}^{\text{dense}} = \{G_1, G_2, \ldots\}$ are ordered from west to east. G_j's are processed in order to produce the output set $\mathcal{G}^{\text{greedy}}$. If G_j can be accommodated by enlarging the rectangle G that was last produced in $\mathcal{G}^{\text{greedy}}$ without introducing false positives (i.e., $\text{MEB}(G, \text{snap}(G_j)) \cap (\mathcal{S} \setminus \mathcal{G}^{\text{dense}}) = \emptyset$, where MEB denotes minimum enclosing box), G is replaced by $\text{MEB}(G, \text{snap}(G_j))$. Otherwise, $\text{snap}(G_j)$ (if it is not $\emptyset$) is added to $\mathcal{G}^{\text{greedy}}$. Thanks to the special properties of rectangle sets produced by *Paint-Dense*, it can be shown that this greedy approach, and *Paint-Sparse* as a whole, is within a factor of 2 optimal in the number of messages generated for any exact algorithm (without assuming dense subscriptions).

Theorem 16. *Given any set of subscriptions, the number of CN messages generated by* Paint-Sparse *is at most twice the minimum possible for any exact algorithm.*

Proof. Let $\mathcal{G}^{\text{dense}} = \{G_1, G_2, \ldots\}$ and $\mathcal{P} = \mathcal{G}^{\text{dense}} \cap \mathcal{S}$. Let $\mathcal{P}_a = \mathcal{P} \cap G_a$. If a rectangle $G \in \mathcal{G}^{\text{opt}}$ contains points of $\mathcal{P}_a$ and $\mathcal{P}_b$ for $1 \leq a < b \leq k$, then G also covers $\mathcal{P}_{a+1}, \ldots, \mathcal{P}_{b-1}$. This property implies that the greedy algorithm is 2-approximate because each G is covered by one or two rectangles generated by *Paint-Sparse*. By construction, *Paint-Sparse* skips all uninteresting exposed objects. Therefore, the number of messages generated by *Paint-Sparse* is at most twice the minimum possible for any exact algorithm. □

The cost of each greedy step is dominated by the test of whether G can accommodate G_j. This test can be done by evaluating a small constant number of snap queries.[7] Since $|\mathcal{G}^{\text{dense}}| \leq k$, *Paint-Sparse* spends $O(k \log m)$ time to generate messages for the updated object and for each exposed object.

An exposed object h is said to be *interesting* if the gain in h's influence region contains some subscription in $\mathcal{S}$; i.e., some message(s) must be generated with h's values. To achieve the second goal above of skipping inessential exposed objects without enumerating all exposed objects, *Paint-Sparse* modifies the method of finding the next exposed object as follows. Suppose the sweep is

[7] Specifically, the region $\text{MEB}(G, \text{snap}(G_j)) \setminus G \setminus \text{snap}(G_j)$ is covered with at most 3 rectangles, and check whether snap returns $\emptyset$ for all of them.

currently at position z, where the updated object i's y-value is set to z. Recall that $\mathrm{IR}^z(i)$ and $\mathrm{II}^z(i)$ denote the influence region and influence interval of i at this point. With the knowledge of $\mathcal{S}$, let $\ell^z = \min\{\ell \mid (\ell, r) \in \mathcal{S} \cap \mathrm{IR}^z(i)\}$ and $r^z = \max\{r \mid (\ell, r) \in \mathcal{S} \cap \mathrm{IR}^z(i)\}$; i.e., (ℓ^z, r^z) is the apex of the smallest southeast quadrant $Q^z \subseteq \mathbb{S}$ enclosing all subscriptions in $\mathrm{IR}^z(i)$, as illustrated in Figure 5.11. *Paint-Dense* finds the next exposed object h to process as $h = \min_y(\mathrm{II}^z(i), z)$ in $\mathbb{E}$. However, h is interesting only if its quadrant θ_h intersects with quadrant Q^z containing actual subscriptions. Hence, the next exposed object h to process is found as $h = \min_y(\mathrm{II}^z(i) \cap [\ell^z, r^z], z) = \min_y([\ell^z, r^z], z)$ in $\mathbb{E}$, allowing *Paint-Sparse* to skip inessential exposed objects.

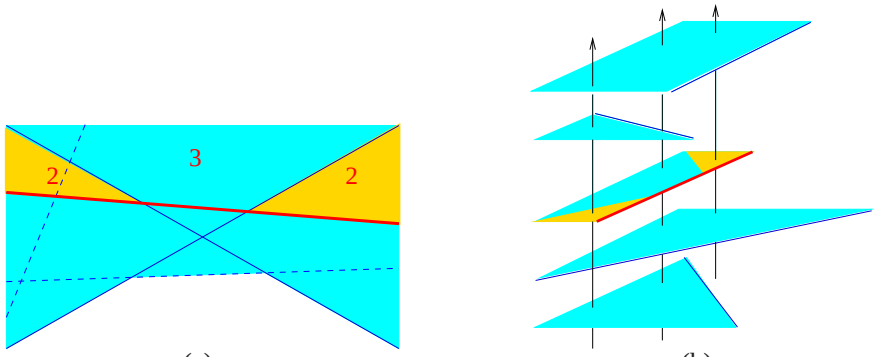
Note that given $\mathcal{L}^z$ and $\mathcal{R}^z$ (see Section 5.4.1), ℓ^z and r^z can be computed from the answers of up to k snap calls in $\mathcal{S}$, one for each south-north rectangle covering $\mathrm{IR}^z(i)$. Thus, compared with *Paint-Dense*, *Paint-Sparse* spends an extra $O(k \log m)$ time for finding each interesting exposed object, and as discussed above, an extra $O(k \log m)$ time to merge messages for the updated object and for each interesting exposed object. The overall time complexity of *Paint-Sparse* is summarized below.

Theorem 17. Paint-Sparse *runs in time $O(t(n)+k \log m)$ for a rank-raising update, and $O(\check{\nu}(t(n)+k \log m))$ time for a rank-lowering update, where $\check{\nu}$ is the number of interesting exposed objects (to distinguish it from ν in Theorem 12, the number of exposed objects). For a rank-lowering update, $\check{\nu} < \check{\mu} \leq (\check{\nu}+1)k$, where $\check{\mu}$ is the number of messages generated by* Paint-Sparse.

Proof. For each exposed object, *Paint-Sparse* also performs snap $O(k)$ queries on the set of subscriptions besides a $\min_y$ query on the set of objects. Therefore, a rank-lowering update requires an additional $O(k \log m)$ cost for each interesting exposed object. The remaining part of the proof follows from the same argument as in the proof of Theorem 12. □

Remark. If the entire $\mathcal{S}$ is too expensive to maintain for the server, it can maintain a small sketch of $\mathcal{S}$, e.g., a cover of $\mathcal{S}$ by B rectangles in $\mathbb{S}$ for a parameter B, and use this cover instead of $\mathcal{S}$ itself in *Paint-Sparse*. This approach would provide a continuous trade-off between the cost of maintaining and utilizing information about subscriptions and the number of CN messages generated.

Paint-Sparse's optimization of merging multiple messages, while reducing the number of messages, increases the areas of rectangles in $\mathbb{S}$ corresponding to messages. Larger areas may, for some CN implementations, imply higher dissemination costs. Nonetheless, note that message merging



(a) (b)

FIGURE 5.12: **a)** Subscription space. Each halfplane is shown in blue color. Suppose the halfplane with red line correspond to object i, and the ones with dashed lines are ranked lower than i. Then $\mathrm{IR}(i)$ is the yellow region when $k = 2$. The numbers indicate the rank of object i for all subscriptions in the cells defined by the solid lines. **b)** Lifted subscription space. $\tilde{\theta}_i$ is the second halfplane intersected by any line $\tilde{\sigma}_i$ passing through the yellow regions in $(+y)$-direction.

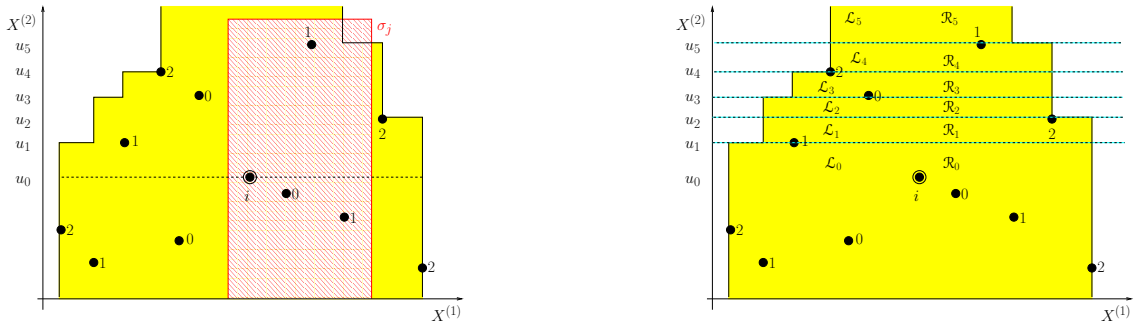
in *Paint-Sparse* is done in a careful way to avoid false positives, so these larger areas do not reach any more subscriptions and traffic to subscriptions remains minimized. Furthermore, reducing the number of messages is effective in relieving the bottleneck at the server and message injection point.

5.5 Generalization

The geometric framework presented in this chapter is quite general and extends to high dimensions as well as different types of ranges and user preferences. First, a user interest needs not be an axis-aligned rectangles; it can be a disk, halfplane, etc. Second, $\mathbb{E}$ can be generalized to a $\mathbb{R}^d$; α of the d attributes are used for *selection* by subscriptions. Although the objects are ranked w.r.t. a common attribute Y in this chapter, additional β parameters can be specified for the ranking function. For example, in Chapters 3 and 4, a user preference is defined as a linear combination of d attribute weights, and $\beta = d - 1$. In this generalized setting, $\mathbb{S}$ and $\tilde{\mathbb{S}}$ becomes $\mathbb{R}^{\alpha+\beta}$ and $\mathbb{R}^{\alpha+\beta+1}$, respectively.

5.5.1 Halfplane query

This section shows how to handle selection ranges which are specified as linear constraints. For example, if a real estate buyer is looking for houses whose carpet-to-saleable area ratio is at least 80%,



(a) (b)

FIGURE 5.13: Influence rectilinear polygon $\mathrm{IP}(i) \subset \mathbb{E}$ for $k = 3$. **a)** Only the objects which are ranked higher than object i are displayed. The number next to each object i' indicates the number of objects in the range $(\min\{x_i^{(1)}, x_{i'}^{(1)}\}, \max\{x_i^{(1)}, x_{i'}^{(1)}\}) \times (-\infty, \max\{x_i^{(2)}, x_{i'}^{(2)}\})$. **b)** Partitioning $\mathrm{IP}(i)$ into rectangles.

her selection range can be expressed as the linear constraint: $carpet\ area - 0.8 * saleable\ area > 0$. This range query is also known as a *halfplane query* when $\mathbb{E} = \mathbb{R}^2$. To maintain top-$k$ objects for the case of halfplane, each subscription $\sigma_j : x^{(2)} \geq a_j^{(1)} x^{(1)} + a_j^{(2)}$ is mapped to a point $\sigma_j^* = (a_j^{(1)}, a_j^{(2)})$ in the subscription space $\mathbb{S}$; and each object i is mapped to a halfplane $\theta_i : \chi^{(2)} \leq -x_i^{(1)} \chi^{(1)} + x_i^{(2)}$ in $\mathbb{S}$. To rank the set of objects for each subscription, each halfplane θ_i is further mapped to $\theta_i \times [y_i, \infty)$ in the lifted subscription space $\tilde{\mathbb{S}}$. It can be verified that S_j is interested in i iff $\sigma_j^* \in \theta_i$. Figure 5.12 shows an example of the influence region $\mathrm{IR}(i) \subseteq \theta_i$ for the case of halfplane. By triangulating an arrangement of halfplanes, $\mathrm{IR}(i)$ can always be partitioned into a set of triangles, each of which can be described using $O(1)$-size. When the y-value of object i is updated, triangle messages (instead of rectangle ones) are generated and inserted into the network.

5.5.2 1.5-dimensional range subscriptions

As a concrete illustration of how the framework is generalized to a higher dimensional case, this section presents algorithms for 1.5-dimensional range subscriptions in $\mathbb{E} = \mathbb{R}^3$. Two numeric attributes $\{X_1, X_2\}$ are used for selection by subscriptions, and an additional numeric attribute Y is used for ranking (in ascending order). Each object $i \in \mathcal{O}$ is modeled as a point $(x_i^{(1)}, x_i^{(2)}, y_i) \in \mathbb{E}$. Each subscription $S_j \in \mathcal{S}$ specifies a region of interest $\sigma_j = [\ell_j^{(1)}, r_j^{(1)}] \times (-\infty, r_j^{(2)}] \subseteq \mathbb{R}^2$. An

Algorithm 6: *ComputeInfluenceRectilinearPolygon*($x_i^{(1)}$, $x_i^{(2)}$, y_i)

begin
  $t \leftarrow 0$; $u_t \leftarrow x_i^{(2)}$;
  $\mathcal{L}_t \leftarrow \text{first}_k(x_i^{(1)}, x_i^{(2)}, y_i, \leftarrow)$; $\mathcal{R}_t \leftarrow \text{first}_k(x_i^{(1)}, x_i^{(2)}, y_i, \rightarrow)$;
  $\text{II}_t = \text{conv}(\mathcal{L}_t \cup \mathcal{R}_t)$; $h_j \leftarrow \min_{X^{(2)}}(\text{II}_t, x_i^{(2)}, y_i)$;
  while $h_j \neq \emptyset$ **do**
    $(\mathcal{L}_{t+1}, \mathcal{R}_{t+1}) \leftarrow$ *UpdateList*$(\mathcal{L}_t, \mathcal{R}_t, x_i^{(1)}, x_{h_j}^{(1)})$;
    $\text{II}_{t+1} = \text{conv}(\mathcal{L}_{t+1} \cup \mathcal{R}_{t+1})$; $u_{t+1} \leftarrow x_{h_j}^{(2)}$;
    $h_j \leftarrow \min_{x^{(2)}}(\text{II}_{t+1}, u_{t+1}, y_i^{\text{new}})$;
    $t \leftarrow t + 1$;
  return $(\mathcal{L}_z, \mathcal{R}_z, u_z, \text{II}_z)_{z=0}^{t}$;
end

example is shown in Figure 5.13(a).

Recall that for the single dimensional range subscriptions, the influence interval of object i, $\text{II}(i)$, contains σ_j if $i \in \text{top}_k(S_j)$. For 1.5-dimensional range subscriptions, $\text{II}(i)$ is generalized to be an *influence (rectilinear) polygon* $\text{IP}(i) \subset \mathbb{E}$. More precisely, $\text{IP}(i)$ is defined by the left and right X_2-monotone boundary chains, as shown in Figure 5.13(a). For any point $(h^{(1)}, h^{(2)})$ on the left boundary chain of $\text{IP}(i)$, $h^{(1)}$ is the same as $x_{i'}^{(1)}$, where i' is the k-th rightmost object in $\mathcal{O}$ in the orthant $\{(x^{(1)}, x^{(2)}, y) \in \mathbb{E} \mid x^{(1)} \leq x_i^{(1)}, x^{(2)} \leq \max\{x_i^{(2)}, h^{(2)}\}, y \leq y_i\}$. The right boundary chain can be defined similarly. As in the case of $\text{II}(i)$, object i is contained in subscription S_j's top-k list only if $\sigma_j \subseteq \text{IP}(i)$. For instance, in Figure 5.13(a), object i ranks below k for subscription σ_j as σ_j is not contained in $\text{IP}(i)$.

Algorithm 6 shows the sweep plane algorithm for computing $\text{IP}(i)$. A plane is swept across the input objects from $x^{(2)} = x_i^{(2)}$ to $x^{(2)} = \infty$ in $\mathbb{E}$. For each $x^{(2)}$-value $u \in [x_i^{(2)}, \infty)$, $\mathcal{L}$ maintains the first k objects in $\mathcal{O}$ in the orthant $\{(x^{(1)}, x^{(2)}, y) \in \mathbb{E} \mid x^{(1)} \leq x_i^{(1)}, x^{(2)} \leq u, y \leq y_i\}$ when proceeding in the (-$x^{(1)}$)-direction. Similarly, $\mathcal{R}$ maintains the first k objects in $\mathcal{O}$ in the orthant $\{(x^{(1)}, x^{(2)}, y) \in \mathbb{E} \mid x^{(1)} \geq x_i^{(1)}, x^{(2)} \leq u, y \leq y_i\}$ when proceeding in the ($x^{(1)}$)-direction. One key observation is that $\mathcal{L}$ or $\mathcal{R}$ is changed only if the plane crosses an object $i' \in \text{IP}(i)$ which is ranked higher than object i, i.e., $y_{i'} < y_i$. Therefore, $\text{IP}(i)$ can be presented in the form of $(\mathcal{L}_z, \mathcal{R}_z, u_z)_{z=0}^{t}$ by partitioning it into a set of rectangles, as shown in Figure 5.13(b). Let $\mathcal{L}_z$ and $\mathcal{R}_z$ be the current lists. When the sweep plane crosses object i', the lists are updated and stored as

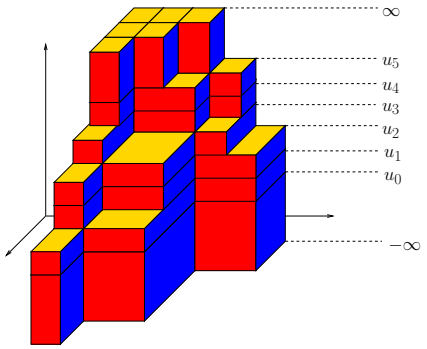


FIGURE 5.14: Influence region in $\mathbb{S}$; $k = 3$.

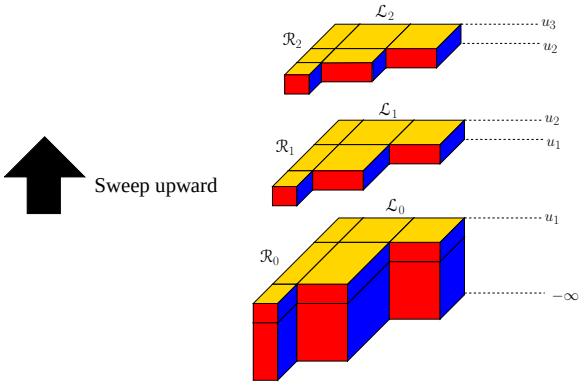


FIGURE 5.15: Computing the influence region in $\mathbb{S}$.

$\mathcal{L}_{z+1}$ and $\mathcal{R}_{z+1}$; u_{z+1} is set to $x_{i'}^{(2)}$.

The subscription space $\mathbb{S}$ is generalized to $\mathbb{R}^3$, where each subscription S_j is mapped to the point $(\ell_j^{(1)}, r_j^{(1)}, r_j^{(2)})$. Object i is mapped to the orthant in $\mathbb{S}$ with apex at $(x_i^{(1)}, x_i^{(1)}, x_i^{(2)})$, i.e., $\{(\xi^{(1)}, \xi^{(2)}, \xi^{(3)}) \in \mathbb{E} \mid \xi^{(1)} \leq x_i^{(1)}, \xi^{(2)} \geq x_i^{(1)}, \xi^{(3)} \geq x_i^{(2)}\}$. Subscription S_j is interested in object i iff $(\ell_j^{(1)}, r_j^{(1)}, r_j^{(2)})$ is contained in the orthant with apex at $(x_i^{(1)}, x_i^{(1)}, x_i^{(2)})$. For 1.5-dimensional range subscriptions, the influence region of i, $\mathrm{IR}(i) \subset \mathbb{S}$, is a rectilinear polyhedron whose vertices are determined by the objects who are ranked higher than i in $\mathrm{IP}(i)$; see Figure 5.14. Given $(\mathcal{L}_z, \mathcal{R}_z, u_z)_{z=0}^t$, a set of tiles can be computed to precisely describe $\mathrm{IR}(i)$; see Figure 5.15.

5.5.2.1 Rank-raising update

Recall that for the single dimensional range subscriptions, given a rank-raising update of object i, the two lists $\mathcal{L}^{\mathrm{new}}$ and $\mathcal{R}^{\mathrm{new}}$ are computed by $\mathrm{first}_k(x_i, y_i^{\mathrm{new}}, \leftarrow)$ and $\mathrm{first}_k(x_i, y_i^{\mathrm{new}}, \rightarrow)$. Given $\mathcal{L}^{\mathrm{new}}$ and $\mathcal{R}^{\mathrm{new}}$, the influence region $\mathrm{IR}^{\mathrm{new}}(i)$ can then be computed in time linear in the number of vertices of $\mathrm{IR}^{\mathrm{new}}(i)$. Now for 1.5-dimensional range subscriptions, the sweep line algorithm returns $(\mathcal{L}_z^{\mathrm{new}}, \mathcal{R}_z^{\mathrm{new}}, u_z^{\mathrm{new}})_{z=0}^t$, as discussed above. For each $z \in \{0, 1, \cdots, t\}$, messages are generated in the same way as the case of 1-dimensional range subscriptions, except that each rectangle message is 3-dimensional (the extra side is $(u_z, u_{z+1}]$). These rectangles together contains all and only the set of affected subscriptions whose $x^{(2)}$-value is between $(u_z, u_{z+1}]$. Note that messages generated at different z values may be compatible with each other,

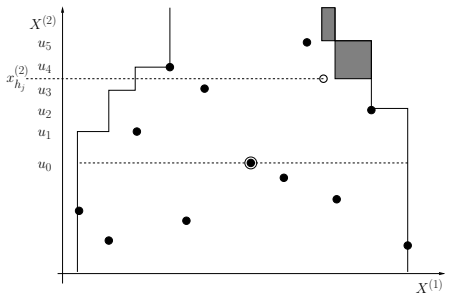


FIGURE 5.16: The new exposed object h_j is shown as a circle. The influence rectilinear polygon is shrunk by pruning the shaded region.

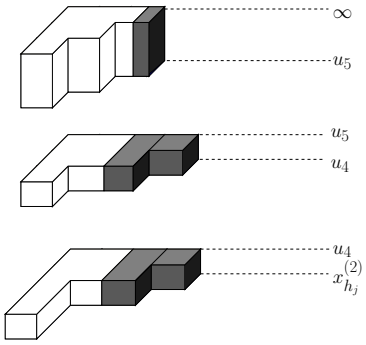


FIGURE 5.17: Subscriptions in the shaded regions need to be notified of h_j. The two messages at the bottom can be further merged with the two messages in the middle.

i.e., $\mathrm{MEB}(\mathsf{Msg}_1, \mathsf{Msg}_2) = \mathsf{Msg}_1 \cup \mathsf{Msg}_2$. They can be further merged to reduce the number of messages to tile the influence region $\mathrm{IR}^{\mathrm{new}}(i)$.

5.5.2.2 Rank-lowering update

Given the rank-lowering update of object i, the algorithm first computes the influence region $\mathrm{IR}^{\mathrm{old}}(i)$. All the subscriptions in the influence region are notified of object i's new y-value. In order to compute the new k-th ranked objects for the set of affected subscriptions, the algorithm needs to sweep the y-value of i continuously from y_i^{old} to y_i^{new} to find the set of exposed objects (as in the case of the single dimensional range subscriptions). That is, the algorithm nestedly sweeps along two dimensions—$x^{(2)}$ and y. When sweeping from y_i^{old} to y_i^{new}, if an exposed object h_j is found in $\mathrm{IP}(i)$, the algorithm updates $\mathrm{IP}(i)$ in $\mathbb{E}$ and $\mathrm{IR}(i)$ in $\mathbb{S}$ by sweeping the plane along the $x^{(2)}$-dimension. Figures 5.16 and 5.17 illustrate the updates at critical time $y = y_{h_j}$. The plane is swept from $x^{(2)} = x^{(2)}_{h_j}$ to $x^{(2)} = q$, where q is the minimum $x^{(2)}$-value such that $x^{(1)}_{h_j} \neq \mathrm{conv}(\mathcal{L} \cup \mathcal{R})$. That is, if a ray is shot through the point $(x^{(1)}_{h_j}, x^{(2)}_{h_j}, y_{h_j})$ in $x^{(2)}$ direction, it hits the boundary of $\mathrm{IP}(i)$ at $x^{(2)} = q$. The algorithm generates messages only for those subscriptions that must receive the object h_j.

5.5.3 *Range Conditions in Higher Dimensions*

For simplicity, only one attribute Y is used for ranking for now; this constraint will be removed later. The event space $\mathbb{E}$ is now $\mathbb{R}^{d+1}$ ($\alpha = d$), and each object i is represented as a point $(x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(d)}, y_i) \in \mathbb{E}$. Each subscription S_j specifies a region $\sigma_j \subseteq \mathbb{R}^d$, which can be a d-dimensional box, halfspace, ball, or simplex, or any other shape, and contains the top-k objects among the ones in which it is interested. Each subscription is mapped to a point σ_j^* and each object i to a region θ_i in the subscription space $\mathbb{S}$ so that S_j is interested in object i iff $\sigma_j^* \in \theta_i$. The exact mapping depends on the shape of subscriptions. If each σ is a d-dimensional box $\prod_{h=1}^{d}[\ell_j^{(h)}, r_j^{(h)}]$, then $\mathbb{S} = \mathbb{R}^{2d}$, $\sigma_j^* = (\ell_j^{(1)}, r_j^{(1)}, \cdots, \ell_j^{(d)}, r_j^{(d)})$, and θ_i is the orthant $\{\xi \in \mathbb{R}^{2d} \mid \xi^{(2i-1)} \leq x_i, \xi^{(2i)} \geq x_i, 1 \leq i \leq d\}$. If each σ_j is a halfplane $x^{(d)} \geq a_j^{(1)}x^{(1)} + \cdots + a_j^{(d-1)}x^{(d-1)} + a_j^{(d)}$, then $\mathbb{S} = \mathbb{R}^d$, $\sigma_j^* = (a_j^{(1)}, \ldots, a_j^{(d)})$, and θ_i is a halfspace $\xi^{(d)} \leq -x_i^{(1)}\xi^{(1)} - \cdots - x_i^{(d-1)}\xi^{(d-1)} + x_i^{(d)}$. If $d = 2$ and each σ_j is a disk of radius r_j centered at (a_j, b_j), then $\mathbb{S} = \mathbb{R}^3$, $\sigma_j^* = (a_j, b_j, a_j^2 + b_j^2 - r_j^2)$ and θ_j is the halfspace $\xi^{(3)} \leq 2x_i^{(1)}\xi^{(1)} + 2x_i^{(2)}\xi^{(2)} - x_i^{(1)} - x_i^{(2)}$. It can be verified that, in each case, S_j is interested in i iff $\sigma_j^* \in \theta_i$. The notion of influence region $\mathrm{IR}(i) \subseteq \theta_i$ can be extended to high dimensions. When the y-value of an object i is updated, $\mathrm{IR}(i)$ is updated from $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ (if y_i is increased) into constant-size regions, and send one $O(1)$-size message for each such region. Computing the decomposition of $\mathrm{IR}^{\mathrm{old}}(i) \setminus \mathrm{IR}^{\mathrm{new}}(i)$ or $\mathrm{IR}^{\mathrm{new}}(i)$ becomes more challenging and the number of regions increases, typically exponentially in the worst case, with dimension. However, many of these regions are empty, so *Paint-Sparse* is more effective in high dimensions. In many cases, it is possible to analyze the number of messages generated by the algorithm. The theorem below gives such a result for the case of rectangles.

Theorem 18. *If the input objects are i.i.d. in $\mathbb{R}^d$ with their attributes being independent and each subscription is an axis-aligned rectangle, then* Paint-Dense *generates $O((k \ln^{d-1} n)^{d+1})$ expected number of CN messages to process an update event.*

Proof. Let $\mathbb{H} \subset \mathbb{E}$ be the hyperplane normal to the d^{th} dimension of $\mathbb{E}$. let i' be the projection of object i onto $\mathbb{H}$. An object j is dominated by another object k with respect to object i iff $k' \in \mathrm{MEB}(j', i') \in \mathbb{H}$ and k ranks higher than j. Let $\mathcal{U}$ denote the set of objects dominated by k

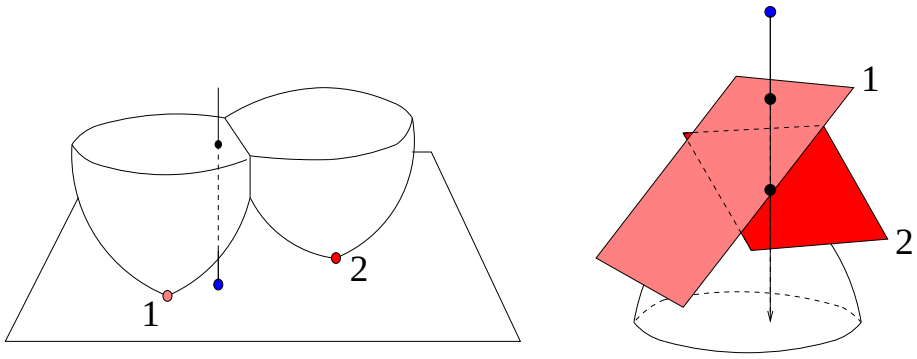


FIGURE 5.18: Lifting transform for k nearest neighbor query in $\mathbb{R}^2$. **a)** The blue query object is closer to object 1 than object 2 in $\mathbb{E}$. **a)** Object hyperplanes are lifted in $\tilde{\mathbb{S}}$. They are tangent to an upside-down paraboloid. The query ray hits hyperplane 1 before hyperplane 2 in $(-\xi^{(d+1)})$-direction.

objects with respect to object i. In $\mathbb{S}$, the influence region of i, $\mathrm{IR}(i)$, is defined by the k-skyband[8] with respect to i.

The average size of the skyline for a set of i.i.d. points is $\theta(\ln^{d-1} n/(d-1)!)$ if attributes are uncorrelated [31]. Under the same assumption, $\mathrm{IR}(i)$ is covered by $\Theta(k \ln^{d-1} n/(d-1)!)$ orthants. The authors in [72] prove that if the number of "octants" to cover an influence region in $\mathbb{R}^{2d}$ is z, then the total number of rectangles for partitioning the influence region will be z^d in the worst case.

Hence, $\mathrm{IR}(i)$ can be partitioned into $\Theta((k \ln^{d-1} n/(d-1)!)^d)$ rectangle messages. Thus, $\Theta((k \ln^{d-1} n/(d-1)!)^d)$ rectangle messages are needed for a modified object i and each exposed object.

The influence region of i, $\mathrm{IR}(i)$, is a rectilinear polyhedron in $\mathbb{S}$ whose vertices are defined by the objects in $\mathcal{U}_k(i)$.

Using the same argument in the proof of Lemma 14, the expected number of objects injected by the rank-lowering update is at most $(k \ln^{d-1} n/(d-1)!)$. This completes the proof for the upper bound. The lower bound construction in Figure 5.9 can be extended for high dimension. □

[8] The k-skyband is the set of objects dominated by at most k objects.

5.5.4 *Combination of range conditions and user preferences*

Each user may specify her preference using β parameters. For example, a preference function can be defined as a linear combination of $\beta + 1$ attributes, as defined in Chapters 3 and 4. As another example, when searching for a gas station, a preference function can be a k nearest neighbor query in $\mathbb{R}^\beta$. By a standard lifting argument [56], each object $(y_i^{(1)}, y_i^{(2)}, \cdots, y_i^{(\beta)})$ can be mapped to the dual plane

$$\xi^{(\beta+1)} = 2y_i^{(1)}\xi^{(1)} + 2y_i^{(2)}\xi^{(2)} + \cdots + 2y_i^{(\beta)}\xi^{(\beta)} - [(y_i^{(1)})^2 + (y_i^{(2)})^2 + \cdots (y_i^{(\beta)})^2],$$

as shown in Figure 5.18. The k nearest neighbors in $\mathcal{O}$ to a query object $(y_i^{(1)}, y_i^{(2)}, \cdots, y_i^{(\beta)})$ correspond to the first k dual hyperplanes intersected by the line $\{(y_i^{(1)}, y_i^{(2)}, \cdots, y_i^{(\beta)}, t) \mid t \in \mathbb{R}\}$ in $(-\xi^{(d+1)})$-direction.

Suppose α attributes are used for selection by subscriptions. Let $(x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(\alpha)}$ be the coordinates of object i for those attributes. By combining range conditions and user preferences, each object has $\alpha+\beta$ parameters $\{x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(\alpha)}, y_i^{(1)}, y_i^{(2)}, \ldots, y_i^{(\beta)}\}$. It is mapped to a region θ_i in the subscription space $\mathbb{S} = \mathbb{R}^{\alpha+\beta}$. Each subscription is mapped to a point in the subscription space $\mathbb{S} = \mathbb{R}^{\alpha+\beta}$. Again, the lifted subscription space $\tilde{\mathbb{S}} = \mathbb{S} \times \mathbb{R}$ can be used to capture the ranking of objects.

5.6 Extensions

5.6.1 *Batch Processing*

For some applications, events can be batched; subscriptions only need to have their top-k lists correctly updated at the end of the batch. Processing events in batched sequence $\mathcal{E}$ one at a time would be an overkill: enough messages would be sent such that each subscription S_j can construct all intermediate states of $\text{top}_k(S_j)$ during $\mathcal{E}$. Given that only the final state of $\text{top}_k(S_j)$ is needed at the end of $\mathcal{E}$, we want to minimize the number of messages delivered to the subscriptions. An algorithm *Paint-Batch* is developed to achieve this goal within the problem setting of Section 5.2.1 without assuming new dissemination interfaces or capabilities. To process individual events, *Paint-Batch* can use any algorithm $\mathcal{A}$ (either *Paint-Dense* or *Paint-Sparse*), with only minor

modifications. *Paint-Batch* itself does not assume the knowledge of $\mathcal{S}$ (though the version using *Paint-Sparse* uses $\mathcal{S}$ indirectly).

The key idea is to pre-process $\mathcal{E}$ in a way such that event-at-time processing by $\mathcal{A}$ (with some modifications) will minimize the number of messages subscriptions receive. Let $\mathrm{IR}^{\mathrm{old}}(i)$ (resp. $\mathrm{IR}^{\mathrm{new}}(i)$) denote the influence region of object i before (resp. after) $\mathcal{E}$. *Paint-Sparse* proceeds in four steps:

1. ***Pre-process.*** First, if multiple events in $\mathcal{E}$ update the same object, they are coalesced into one. More precisely, if $\mathcal{E}$ contains the sequence $\mathsf{Upd}(x_i, y_i^{(0)} \to y_i^{(1)}), \ldots, \mathsf{Upd}(x_i, y_i^{(c-1)} \to y_i^{(c)})$, they are replaced with a single $\mathsf{Upd}(x_i, y_i^{(0)} \to y_i^{(c)})$. Next, the set $\mathcal{E}$ is split into two, $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$, where $\mathcal{E}^{\downarrow}$ (resp. $\mathcal{E}^{\uparrow}$) contains all events that decrease (resp. increase) y-value.

2. ***Apply*** $\mathcal{E}^{\downarrow}$ ***to*** $\mathcal{O}$**.** Let $\mathcal{T}$ denote the data structure that is maintained for $\mathcal{O}$. $\mathcal{T}$ is updated with using events in $\mathcal{E}^{\downarrow}$. No message is generated in this step.

3. ***Generate messages for*** $\mathcal{E}^{\uparrow}$ ***and apply*** $\mathcal{E}^{\uparrow}$ ***to*** $\mathcal{O}$**.** Events are processed in $\mathcal{E}^{\downarrow}$ (all of which are rank-lowering) in descending order of the new values using $\mathcal{A}$, but with the following modifications. 1) If $\mathcal{A}$ generates messages for an exposed object that is updated in $\mathcal{E}^{\uparrow}$ or will be later updated in $\mathcal{E}^{\downarrow}$, such messages are discarded and would not be sent. 2) If $\mathcal{A}$ is processing a ranking-lowering update for an object i whose messages have been discarded earlier, instead of notifying the region $\mathrm{IR}^{\mathrm{old}}(i)$ with i's updated values as $\mathcal{A}$ would normally do, the algorithm notifies the region $\mathrm{IR}^{\mathrm{new}}(i) \cup \mathrm{IR}^{\mathrm{pre}}(i)$, where $\mathrm{IR}^{\mathrm{pre}}(i)$ denotes i's influence region right before the algorithm starts processing $\mathcal{E}^{\uparrow}$. To implement these modifications, there is no need to remember all $\mathrm{IR}^{\mathrm{pre}}(i)$'s, which would require $\Theta(nk)$ space. It turns out that it is sufficient to maintain an $O(n)$-space data structure so that $\mathrm{IR}^{\mathrm{pre}}(i)$ can be computed on demand, without increasing the time complexity of $\mathcal{A}$. More specifically, besides the data structure $\mathcal{T}$ normally maintained for $\mathcal{O}$, an additional data structure $\mathcal{T}'$ is maintained to index the set of objects updated in $\mathcal{E}^{\uparrow}$. $\mathcal{T}'$ is initially empty before the algorithm starts processing $\mathcal{E}^{\uparrow}$. When processing $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \to y_i^{\mathrm{new}}) \in \mathcal{E}^{\uparrow}$ in the current iteration, in addition to updating the y-value of object i in $\mathcal{T}$ to y_i^{new}, $(x_i, y_i^{\mathrm{old}})$ is inserted into $\mathcal{T}'$. While computing $\mathrm{IR}^{\mathrm{new}}(i)$ uses $\mathcal{T}$, computing $\mathrm{IR}^{\mathrm{new}}(i) \cup \mathrm{IR}^{\mathrm{pre}}(i)$ uses $\mathcal{T}$ and $\mathcal{T}'$.

4. ***Generate messages for*** $\mathcal{E}^{\downarrow}$**.** For each object i updated in $\mathcal{E}^{\downarrow}$, the region $\mathrm{IR}^{\mathrm{new}}(i)$ is notified with i's new value. The algorithm simply follows $\mathcal{A}$ to compute the messages by querying $\mathcal{T}$.

Below gives some intuition behind the design of *Paint-Batch*.

- *Why does the algorithm generate message for* $\mathcal{E}^{\downarrow}$ *(Step 4) after* $\mathcal{E}^{\uparrow}$ *(Step 2)?* Suppose that an object i is unchanged by $\mathcal{E}$, and $i \in \mathrm{top}_k(S_j)$ both before and after $\mathcal{E}$ for some subscription S_j; in this case, we do not want to notify S_j with i. However, some objects updated in $\mathcal{E}^{\downarrow}$ may temporarily lower the rank of i to below k, before some other objects updated in $\mathcal{E}^{\uparrow}$ raise the rank of i to within k again. Sending messages generated for $\mathcal{E}^{\downarrow}$ before $\mathcal{E}^{\uparrow}$ would cause S_j to drop i, forcing us to notify S_j with i later when processing $\mathcal{E}^{\uparrow}$. Deferring messages for all rank-raising updates avoids this problem.

  Also, if $\mathcal{E}^{\downarrow}$ is processed before $\mathcal{E}^{\uparrow}$, an update to object i in $\mathcal{E}^{\downarrow}$ would enlarge $\mathrm{IR}(i)$, and updates to other objects in $\mathcal{E}^{\uparrow}$ might further enlarge $\mathrm{IR}(i)$; therefore, the algorithm would need to generate messages involving i every time i is exposed in $\mathcal{E}^{\uparrow}$. Although doing so would not cause subscriptions to receive unnecessary messages, it leads to more messages compared with the presented approach, which guarantees that for each updated object i, messages involving i are only generated once (when the algorithm processes the event updating i).

- *Why are the modifications to* $\mathcal{A}$ *necessary when processing* $\mathcal{E}^{\uparrow}$ *(Step 3)?* Suppose the algorithm is currently processing an update that exposes object i, causing it to enter $\mathrm{top}_k(S_j)$ for some subscription S_j at this point. If i will be later updated in $\mathcal{E}^{\uparrow}$, it is possible that i will leave $\mathrm{top}_k(S_j)$ at that point. Without the modifications, S_j would be notified with i unnecessarily.

  On the other hand, if i is updated in $\mathcal{E}^{\downarrow}$, then the gains in $\mathrm{IR}(i)$ during the processing of $\mathcal{E}^{\uparrow}$ should be ignored, because they will covered by $\mathrm{IR}^{\mathrm{new}}(i)$ when i is processed in Step 4.

- *Why does the algorithm process* $\mathcal{E}^{\uparrow}$ *in sorted order (Step 3)?* Processing $\mathcal{E}^{\uparrow}$ in descending order of the new values means that once the algorithm processes an event updating object i in $\mathcal{E}^{\uparrow}$, i will never be exposed again. With this property, for each updated object i, messages involving it are only generated once (when its update event is processed). Without this

property, the algorithm may need to generate messages involving i every time when i is exposed after it is updated. Although doing so would not cause subscriptions to receive unnecessary messages, it may lead to more messages compared with the presented approach.

- *Why does the algorithm need to apply $\mathcal{E}^{\downarrow}$ to $\mathcal{O}$ (Step 2) before processing $\mathcal{E}^{\uparrow}$?* Without applying $\mathcal{E}^{\downarrow}$ to $\mathcal{O}$, the algorithm would essentially process $\mathcal{E}^{\uparrow}$ followed by $\mathcal{E}^{\downarrow}$. It is possible for $\mathcal{E}^{\uparrow}$ to expose an object i, causing it to temporarily enter $\mathrm{top}_k(S_j)$ for some subscriptions; however, $\mathcal{E}^{\downarrow}$ may subsequently make i leave $\mathrm{top}_k(S_j)$. Notifying S_j with i would be unnecessary. Applying $\mathcal{E}^{\downarrow}$ to $\mathcal{O}$ before processing $\mathcal{E}^{\uparrow}$ (in conjunction with way the algorithm processes $\mathcal{E}^{\uparrow}$) ensures that when processing each update in $\mathcal{E}^{\uparrow}$, every object i exposed by this update will remain in the final $\mathrm{top}_k(S_j)$ for every S_j that receives i.

The remainder of this subsection is devoted to prove that each subscriber receives the minimum number of messages possible under *Paint-Batch*.

Lemma 19. *When an event about object i is processed, $\mathrm{IR}^{\mathrm{current}}(i)$ ($\mathrm{IR}^{\mathrm{pre}}(i)$) is the minimum set of subscriptions in $\mathrm{IR}^{\mathrm{old}}(i)$ that must be notified in order to produce the correct final top-k lists for those subscriptions if the event is a rank-raising update (rank-lowering update) for object i.*

Proof. If the event is a rank-raising update for object i, messages are generated after all events in $\mathcal{E}$ have been processed. Hence, the union of the messages for object i is exactly $\mathrm{IR}^{\mathrm{new}}(i)$, in which every subscription needs to be notified. If the event is a rank-lowering update for object i, processing $\mathcal{E}^{\downarrow}$ first guarantees that all events that can shrink $\mathrm{IR}(i)$ have been processed, therefore, all the subscriptions in $\mathrm{IR}^{\mathrm{pre}}(i)$ must be notified about the update of object i no matter how the events in $\mathcal{E}$ is ordered. □

Lemma 20. *If an object $i \in \mathrm{top}_k^{\mathrm{final}}(S_j)$, i is never forced out of S_j's top-k list because of space constraint.*

Proof. First, when a message about object i is generated, all the other objects whose old and new values are larger and smaller than object i's new value must have been processed. Hence, if object i belongs to S_j's final top-k list, it must be higher than k-th in S_j's ranking. Second, during a rank-lowering update for object i, a message about object i is first sent to every subscription S_j that has i in its top-k list. Thus, if a message about an exposed object is also sent to S_j, object i

must have dropped to k-th in S_j's ranking. No other objects in S_j's list are forced to be removed because of the arrival of the exposed object. Third, since messages for all rank-raising updates are generated at the end of the batch process, any object that is replaced by a new arrived object i must satisfy one of the following two conditions: 1) it ranks lower than object i in the final top-k list and 2) its value will later become lower than the value of the k-th item in the top-k list due to a rank-raising update and it will re-enter the final top-k list. □

Lemma 21. *Let $\mathcal{L}$ and $\mathcal{L}'$ be the lists returned by $\mathrm{first}_k(x_i, y_i^{old}, \leftarrow)$ on $\mathcal{T}$ and $\mathcal{T}'$, respectively. Let $\mathcal{R}$ and $\mathcal{R}'$ be the lists returned by $\mathrm{first}_k(x_i, y_i^{old}, \rightarrow)$ on $\mathcal{T}$ and $\mathcal{T}'$, respectively. Let $\mathcal{L}^* = \{\ell_1 > \ell_2 > \cdots > \ell_k\}$ contain the first k values in $\mathcal{L} \cup \mathcal{L}'$ (padded with $-\infty$ if $|\mathcal{L}| < k$). let $\mathcal{R}^* = \{r_1 < r_2 < \cdots < r_k\}$ contain the first k values in $\mathcal{R} \cup \mathcal{R}'$ (padded with ∞ if $|\mathcal{R}| < k$). $\mathrm{IR}^{\mathrm{pre}}(i)$ is an axis-aligned subregion of the quadrant θ_i, with vertices $(x_i, x_i), (\ell_v, x_i), (\ell_v, r_1), (\ell_{v-1}, r_1), (\ell_{v-1}, r_2), \ldots, (\ell_1, r_{v-1}), (\ell_1, r_v), (x_i, r_v)$ in clockwise order, ignoring degenerate vertices with $-\infty$ or ∞ coordinates.*

Proof. Let $\mathcal{L}^{\mathrm{pre}} = \{\ell_1^{\mathrm{pre}}, \ell_2^{\mathrm{pre}}, \cdots, \}$ and $\mathcal{R}^{\mathrm{pre}} = \{r_1^{\mathrm{pre}}, r_2^{\mathrm{pre}}, \cdots, \}$ be the lists returned by $\mathrm{first}(x_i, y_i^{old}, \leftarrow)$ and $\mathrm{first}_k(x_i, y_i^{old}, \rightarrow)$ right before we start processing $\mathcal{E}^{\uparrow}$. Let $\mathcal{L} = \{\ell_1, \ell_2, \cdots, \}$ and $\mathcal{R} = \{r_1, r_2, \cdots, \}$ be the lists returned by $\mathrm{first}(x_i, y_i^{old}, \leftarrow)$ and $\mathrm{first}_k(x_i, y_i^{old}, \rightarrow)$ for the current event. If $\mathcal{L} = \mathcal{L}^{\mathrm{pre}}$ and $\mathcal{R} = \mathcal{R}^{\mathrm{pre}}$, we are done. Otherwise, all objects in $\mathcal{L}^{\mathrm{pre}} \backslash \mathcal{L}$ and $\mathcal{R}^{\mathrm{pre}} \backslash \mathcal{R}$ must have been modified because no other objects' ranking is raised to force those objects out of $\mathcal{L}$ and $\mathcal{R}$ during $\mathcal{E}^{\uparrow}$. Hence, all objects in $\mathcal{L}^{\mathrm{pre}} \backslash \mathcal{L}$ and $\mathcal{R}^{\mathrm{pre}} \backslash \mathcal{R}$ must be indexed by $\mathcal{T}'$ using their old y-values, and they can be retreived by two first_k calls on $\mathcal{T}'$. □

Lemma 22. *No subscription receives more than one message for the same object i.*

Proof. The coalescing step guarantees that no two events update the same object i. The algorithm also guarantees that no message for an object i will be generated if the value of i will be updated later in the sequence. Messages generated for the update of i completely pack $\mathrm{IR}(i)$ such that every subscription in $\mathrm{IR}(i)$ receives one messsage for object i. After the value of object i has been updated, $\mathrm{IR}(i)$ will never be shrunk since all the remaining events are the rank-lowering updates for other objects. Additional messages are generated for object i only if the rank-lowering update for other events further expand $\mathrm{IR}(i)$. However, these messages only cover the expanded part of $\mathrm{IR}(i)$. Therefore, no subscription receives more than one message for the same object. □

Lemma 23. *If a subscription S_j receives a message for an object i and $(x_i, y_i) \notin \mathrm{top}_k(S_j)$ before the start of batched processing, then $(x_i, y_i) \in \mathrm{top}_k(S_j)$ at the end of batched processing.*

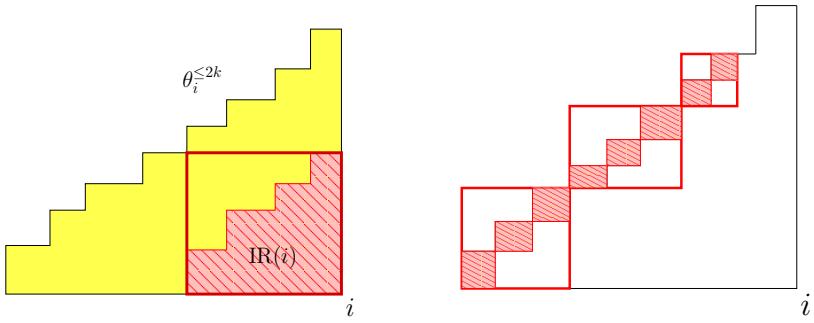
Proof. Assume S_j receives an update for an object i and $(x_i, y_i) \notin \mathrm{top}_k(S_j)$ before the start of batch processing. The only possible way that object i will not be in $\mathrm{top}_k(S_j)$ by the end of batch processing is that $\mathrm{IR}(i)$ will later be shrunk such that it will not contain S_j. There are two cases, in which $\mathrm{IR}(i)$ can be shrunk: the y-value of object i has increased, or the y-value of another object l has decreased. The first case cannot happen since the algorithm does not generate a message for object i if its y-value will be updated later in the sequence. For the second case, since $\mathcal{E}^{\downarrow}$ is processed before $\mathcal{E}^{\uparrow}$, the current update must be a rank-raising update for object i. However, as $\mathcal{E}^{\downarrow}$ is sorted in ascending order of the new values, $y_i^{\mathrm{new}} < y_l^{\mathrm{new}}$, so decreasing the y-value of l has no effect on the rank of i for S_j. □

Theorem 24. Paint-Batch *minimizes the number of messages each subscriber receives. Given an event sequence $\mathcal{E}$,* Paint-Batch *based on* Paint-Dense *runs in $O(|\mathcal{E}| \log |\mathcal{E}| + \bar{\nu} t(n) + \bar{\mu})$ time, and* Paint-Batch *based on* Paint-Sparse *runs in $O(|\mathcal{E}| \log |\mathcal{E}| + \bar{\nu}(t(n) + k \log m))$ time, where $\bar{\mu}$ is the number of messages generated by* Paint-Batch *and $\bar{\nu}$ is the number of objects in these messages.*

Proof. Lemma 19, 22 and 23 together imply that *Paint-Batch* minimizes the number of messages each subscriber receives. *Paint-Batch* requires sorting that takes $O(|\mathcal{E}| \log |\mathcal{E}|)$ time. The other parts of the running times for *Paint-Batch* follow from the same argument as in the proof of Theorem 12 and 17. □

5.6.2 Approximate Algorithms

To further alleviate the potential message injection bottleneck, more reduction in the number of CN messages generated by the server is possible with approximate algorithms. They allow subscriptions to receive unnecessary messages containing false positive updates to top-k lists, which are discarded by post-processing at the subscriptions. The basic idea is to simplify the boundaries of regions to notify by judiciously including some additional subscriptions. As a simple example, Figure 5.19(a) shows that instead of tiling a staircase-shaped $\mathrm{IR}(i)$ with multiple messages, a single message with rectangle $\mathrm{MEB}(\mathrm{IR}(i))$ can be used. Although subscriptions in $\mathrm{MEB}(\mathrm{IR}(i)) \setminus \mathrm{IR}(i)$ would get object i as a false positive, it can be shown that i would still rank within top $2k$ for these



(a) (b)

FIGURE 5.19: Covering regions (shown with dark shade) using fewer rectangles (shown with thick outlines) by allowing false positives. (a) Covering $\mathrm{IR}(i)$ (a staircase) with $\varepsilon = 1$; (b) covering $\mathrm{IR}^{\mathrm{new}}(i) \setminus \mathrm{IR}^{\mathrm{old}}(i)$ (a diagonal chain) with $\varepsilon = \frac{1}{3}$.

subscriptions, because $\mathrm{MEB}(\mathrm{IR}(i)) \subseteq \theta_i^{\leq 2k}$, thanks to the special structures of θ_i^v's established in Section 5.3.

The approximate algorithms, based on *Paint-Dense* and *Paint-Sparse*, generalize this simple but effective idea. They are parameterized by $\varepsilon \in \{\frac{1}{k-1}, \frac{1}{k-2}, \ldots, \frac{1}{2}, 1\}$, which controls the degree of approximation. Consider the task of notifying an ordered list $\mathcal{G}^{\mathrm{dense}}$ of no more than k rectangles (as defined in Section 5.4.2), where $\mathcal{G}^{\mathrm{dense}} \subseteq \mathrm{IR}_{\square}(i)$ for some updated or exposed object i. $\mathcal{G}^{\mathrm{dense}}$ can be divided into no more than $1/\varepsilon$ sublists, such that each sublist contains no more than $\lceil \varepsilon k \rceil$ adjacent rectangles. The rectangles in each sublist are covered by their minimum enclosing box. Figure 5.19(b) shows an example of covering a diagonal chain (representing the gain in some exposed object's influence region) with 3 rectangles ($\varepsilon = \frac{1}{3}$).

Paint-Dense can be made approximate by post-processing each $\mathcal{G}^{\mathrm{dense}}$ as above to generate messages. *Paint-Sparse* can be made approximate by processing $\mathcal{G}^{\mathrm{dense}}$ before subjecting it to greedy message merging. The resulting approximate algorithms are called *Paint-Dense*(ε) and *Paint-Sparse*(ε), respectively. Each subscription S_j follows the same protocol in Section 5.2.1 for maintaining $\mathrm{top}_k(S_j)$. S_j may receive an object that should not enter $\mathrm{top}_k(S_j)$, or one that is already in $\mathrm{top}_k(S_j)$ and has not changed value. Such false positives are automatically ignored by the protocol, and objects in these messages are limited to those ranked around the k-th, as shown by the theorem below. This theorem also shows the reduction in the number of messages and the running times of the approximate algorithms.

Theorem 25. *With the approximate algorithms, a subscription S_j will receive a message with*

object i only if 1) i ranks between $(1-\varepsilon)k$ and $(1+\varepsilon)k$, or 2) i is already in the top $(1+\varepsilon)k$ but its value has changed.

For approximate algorithms, a rank-raising update generates no more than $1/\varepsilon$ messages; a rank-lowering update generates $O(n/\varepsilon)$ messages, with no more than $1/\varepsilon$ per expose object for Paint-Dense(ε) *and no more than $1/\varepsilon$ per interesting expose object for* Paint-Sparse(ε). *If each rank-lowering update chooses an object to update uniformly at random, the expected number of messages generated is $O(k/\varepsilon)$.*

Paint-Dense(ε) *runs in time $O(t(n)+k)$ for a rank-raising update, and $O(\nu t(n)+\hat{\mu}+k)$ time for a rank-lowering update, where ν is the number of exposed objects and $\hat{\mu}$ is the number of messages generated by* Paint-Dense(ε). Paint-Sparse(ε) *runs in time $O(t(n)+k+\log m/\varepsilon)$ for a rank-raising update, and $O(\check{\nu}(t(n)+k+\log m/\varepsilon))$ time for a rank-lowering update, where $\check{\nu}$ is the number of interesting exposed objects.*

Proof. By construction, the top-left and bottom right vertices of each message generated for each exposed object have rank $(1+\epsilon)k$ and $(1-\epsilon)k$, respectively. Hence, any subscription in a message ranks between $(1-\epsilon)k$ and $(1+\epsilon)k$. Similarly, the top-left vertex of a message also ranks $(1+\epsilon)k$ for a modified object i. The proof for the number of messages and the running time follows from the fact that $O(1/\epsilon)$ messages are generated for object i and each expose object and from the same argument as in the proof of Theorem 12 and 17. □

5.6.3 Distributing the database

The central server can be replaced with multiple servers, which together maintain the database of objects in a distributed manner. Recall that objects are mapped to quadrants with apex on the diagonal of the subcription space, $\mathbb{S}$. Suppose there are β servers. The diagonal is partitioned into β zones, and one server is assigned to each zone for maintaining all objects in the zone. Each zone owner maintains pointers to its two immediate (left and right) neighboring zone owners along the diagonal. Since objects are distributed across multiple servers, the set of objects returned by the first_k and $\min_y$ queries may be located at different servers. Consider an event $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \rightarrow y_i^{\mathrm{new}})$. The event is first routed to the server that maintains the object i. Then the two queries $\mathrm{first}_k(x_i, y_i^{\mathrm{new}}, \leftarrow)$ and $\mathrm{first}_k(x_i, y_i^{\mathrm{new}}, \rightarrow)$ are answered in a distributed manner: If the first_k query returns $t < k$ objects on the left (resp. right) side of object i, we traverse to the left (resp. right)

zone-owner and retrieve the remaining $k - t$ objects with a second first_k query. This procedure is repeated until either k objects have been retrieved or all the objects on the left (right) side of i have been examined. Similarly, the set of objects reported by $\min_y$ queries, i.e., the set of objects exposed during a ranking-lowering update, can also be computed by two linear traversals along the diagonal: Let $\sigma' = [\ell', r']$ be the range of objects maintained by the server. Let $\sigma = (\ell, r)$ be the query range. The server computes the object with the minimum y-value in the 3-sided rectangle $\sigma \cap \sigma' \times (y_0, \infty)$. If $\ell < \ell'$ (resp. $r > r'$), we traverse to the left (resp. right) zone owner and repeat the same procedure. The minimum among the returned objects is the answer to the $\min_y$ query. Algorithms 7 and 8 give the pseudo-code for the first_k and $\min_y$ queries.

Algorithm 7: $\mathrm{first}_k(x_i, y_i^{\mathrm{new}}, c)$

```
1  begin
2      if c = ' ←' then
3          L ← first_k(x_i, y_i^new, ←);
4          if |L| < k then
5              L ← L ∪ getFirstKFromLeftNeighbor(x_i, y_i^new, ←, k − |L|);
6          return L;
7      else
8          R ← first_k(x_i, y_i^new, →);
9          if |R| < k then
10             R ← R ∪ getFirstKFromRightNeighbor(x_i, y_i^new, →, k − |R|);
11         return R;
12 end
```

Algorithm 8: $\min_y(\sigma, y_0)$

```
1  begin
2      h_j ← min_y(σ ∩ σ_s, y_0);
3      if ℓ < ℓ_s then
4          h'_j ← getMinYFromLeftNeighbor(σ, y_0);
5          if h'_j < h_j then h_j ← h'_j;
6      if r > r_s then
7          h'_j ← getMinYFromRightNeighbor(σ, y_0);
8          if h'_j < h_j then h_j ← h'_j;
9      return h_j;
10 end
```

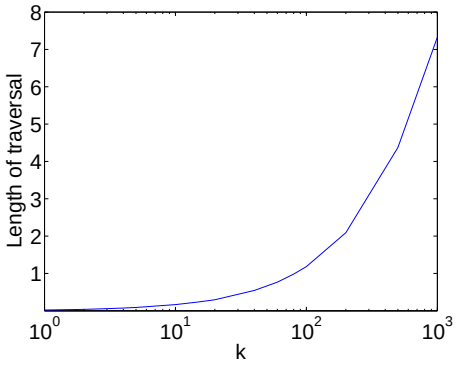


FIGURE 5.20: Length of traversal (100 servers, $100{,}000$ objects).

Suppose each of the β servers maintains $\lceil n/\beta \rceil$ objects. If the input objects are i.i.d. in $\mathbb{R}^2$ with their attributes being independent, the expected number of servers traversed for a first_k or min_y query is roughly $2k\beta/n$. Figure 5.20 shows the empirical results on the average length of a traversal over $10{,}000$ queries when $n = 100{,}000$ and $\beta = 100$.

5.7 Evaluation

Network setup. A CN based on *Meghdoot* [61] and the *content addressable network* [99] is used for message dissemination. This CN uses a network of *brokers* to deliver CN messages of the format described in Section 5.2.1. It partitions the subscription space $\mathbb{S}$ into *zones*, each owned by a broker responsible for all subscriptions within this zone; this broker is called the *gateway* broker of these subscriptions. Each zone can forward messages to its adjacent zones, so messages may travel over multiple hops to their destinations. INET [43] is used to generate a $20{,}000$-node IP network, and randomly pick $1{,}000$ nodes as brokers. Subscriptions are located randomly within the network, and object update events also originate from random locations.

For the approaches presented in this chapter, the broker whose zone covers the center of $\mathbb{S}$ is designated as the server, which maintains the database of all objects $\mathcal{O}$. In the case of sparse subscriptions, the server additionally maintains the database of all subscriptions $\mathcal{S}$ (but not how they are assigned to brokers). Events are first routed to the server, where they are reformulated into a sequence of CN messages.

Approaches compared. The presented approaches all use CN for message dissemination and only differ in their message generation algorithms. Hence, the names of these algorithms are used

to refer to these approaches: exact ones include *Paint-Dense* and *Paint-Sparse*, and approximate ones include *Paint-Dense*(ε) and *Paint-Sparse*(ε) with different ε settings. They are compared with the following approaches, which sample the space of less sophisticated alternatives:

- *Unicast*: An event is first sent to the server, which in this case tracks all objects, all subscriptions, and how subscriptions are assigned to gateway brokers. The server computes the set of affected subscriptions. For each affected subscription S_j, the server unicasts to S_j's gateway broker the id j and the change to $\mathrm{top}_k(S_j)$ (which can be captured by one object). This approach is exact in that it notifies only affected subscriptions.

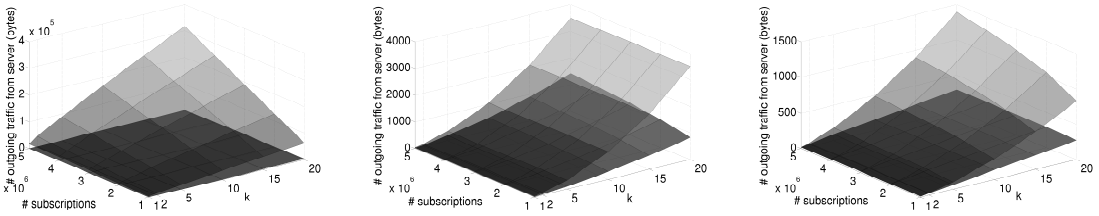
  For comparison, the following algorithm is considered for computing unicast messages, which uses some but not all insights from the presented algorithms.[9] Given $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \to y_i^{\mathrm{new}})$, the server first computes $\mathrm{II}^{\mathrm{old}}(i) \cup \mathrm{II}^{\mathrm{new}}(i) = (\ell, r)$, and finds all subscriptions in $(\ell, x_i] \times [x_i, r) \subseteq \mathbb{S}$. Next, the server processes each such subscription S_j in turn. For a rank-lowering update, the exposed object has y-value between y_i^{old} and y_i^{new}, and can be found by $\mathrm{min}_y(\sigma_j, y_i^{\mathrm{old}})$.

- *CN-Relax:* This approach uses the same CN as the presented approaches, but does not need a server. An event $\mathsf{Upd}(x_i, y_i^{\mathrm{old}} \to y_i^{\mathrm{new}})$ directly enters the CN as $\mathsf{Msg}(x_i, x_i, -\infty, \infty, x_i, y_i^{\mathrm{new}})$, which reaches all subscriptions whose ranges include x_i. In effect, *CN-Relax* treats each range top-k subscription simply as a range subscription. Each subscription must maintain all objects within its range at all times, from which the top k can be computed. This approach is approximate in that it may notify unaffected subscriptions.

Metrics. The following metrics are considered in evaluation:

- *Outgoing traffic from the server*: Measured by the total number of bytes sent by the server. A larger number means higher network stress at the server.

[9] Alternatively, *Paint-Dense* may simply be used to obtain the list of affected tiles in $\mathbb{S}$, and then look up affected subscriptions within these tiles. In this case, the server processing cost becomes that of *Paint-Dense* plus a term linear in the number affected subscriptions, which is strictly (much) less efficient than *Paint-Dense* and does not offer an interesting comparison.



(a) *Unicast* (top) vs. *Paint-Dense* (bottom)

(b) *Paint-Dense* (top) vs. *Paint-Sparse* (bottom)

(c) *Paint-Sparse* (top) vs. *Paint-Sparse*(1) (bottom)

FIGURE 5.21: Average outgoing traffic (# bytes) from server per event.

- *Traffic in the broker network*: Measured by the total number of bytes sent across network hops, excluding those from gateway brokers to their subscriptions (which are accounted for by the *redundancy* metric discussed below). Depending on what is considered as a "hop," there are two metrics: *overlay traffic* treats each overlay link (i.e., a link between two brokers without going through other brokers) as a hop, while *IP traffic* treats each underlying IP link as a hop. IP traffic better reflects physical reality but it depends heavily on the CN implementation; overlay traffic better reflects how the CN is used (as a black box). Well-designed CNs try to make overlay routes as efficient as IP routes, which helps close the gap between these two metrics.

- *Redundancy in messages received by subscriptions*: Measured by $\widehat{N}/N - 1$, where $\widehat{N}$ denotes the number of messages received by subscriptions and N denotes the number of messages received by subscriptions under an exact approach. A larger redundancy means higher last-hop traffic and more work for subscriptions. Exact approaches have 0 redundancy.

- *Server processing cost*: Measured by the number of calls (by type, as discussed in Section 5.2.2) against the underlying data structures when generating messages. This measurement is chosen because the running time depends on the choice of data structures. The implementation uses data structures that are easier to implement and efficient in practice, but not asymptotically optimal.

Workloads. Most results in this section use synthetic workloads, which allow us to vary their characteristics. Unless specified otherwise, there are $10{,}000$ objects, whose x-values follow one of two distributions: 1) *Uniform*: The x-values are uniformly distributed over the possible x-value range. 2) *Clustered*: The x-values lie in 10 clusters, whose centers partition the possible x-value

range into 11 segments of length w. Each cluster gets 10% of the objects. For each object in a cluster, the distance between its x-value and the cluster center follows a Gaussian distribution with standard deviation $w/8$.

To generate an event, an object is picked to update uniformly at random. Its y-value is increased or decreased, each with 0.5 probability. The new y-value is then chosen uniformly random from the possible range of y-values.

Unless specified otherwise, the number of subscriptions is 2 million. The following subscription distributions are considered:

- *Uniform*: The subscriptions are uniformly distributed in $\mathbb{S}$.
- *Clustered*: Most subscriptions lie in 10 clusters in $\mathbb{S}$. Let P be a set of $10{,}000 \times 10{,}000$ grid points. A set C of 10 centers is first randomly picked in $\mathbb{S}$ and use a mixture model to assign probability to each point $p \in P$. A parameter $\triangle$ controls the standard deviation of each cluster $c_i \in \mathcal{C}$. Let σ be $(\max - \min)\triangle/4$, where $\max$ and $\min$ are the maximum and minimum values in the domain. For each point $p \in P$, $F(p) = \sum_{i=1}^{10} F_i(p)$, where $F_i(p) = \exp(-0.5\|c_i - p\|/\sigma^2)$. The probabilities are then normalized such that they sum to 1.
- *Correlated* (to clustered object distribution): Subscriptions are generated from the 10 clusters of the clustered object distribution. For each subscription in a cluster, the distance between its endpoints from the cluster center follows a Gaussian distribution with standard deviation $w/8$.
- *Anti-correlated* (to clustered object distribution): As with the correlated case above, subscriptions are generated using the clusters of the clustered object distribution. However, each cluster center is shifted by $w/2$ and ignore the last cluster, such that each subscription cluster center is located midway between two consecutive object cluster centers for the object distribution.

In addition to synthetic workloads, information on $2{,}031$ stocks have been obtained from Yahoo! Finance. For each stock, its earnings per stock (EPS), the average recommendation (RECO, which varies from 1, strong buy, to 5, strong sell, over the past month), as well as the open and close prices over 30 days, are collected. EPS is then used to convert each price to price-to-earning

ratios (PER). Thus, there is a trace of events, each being an update of PER with a RECO constant. $400{,}000$ subscriptions are generated and each requests the k lowest PER over a RECO range.

5.7.1 Main results

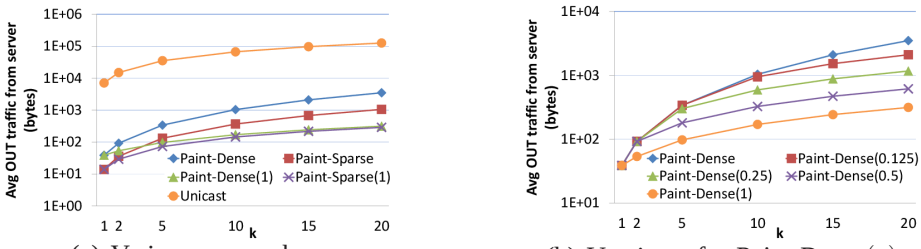
This section will first present results for the uniform object distribution and uniform subscription distribution.

Outgoing traffic from server. Figure 5.21 shows the outgoing traffic from the server per event, averaged over all events in the workload, when varying k and the number of subscriptions (m). For clarity, only two approaches are compared per plot. Note that *CN-Relax* is not compared because it is a serverless approach. Figure 5.21(a) shows that *Paint-Dense*'s outgoing traffic is invariant to m, but *Unicast*'s outgoing traffic is not scalable in m and k. When $m = 5{,}000{,}000$ and $k = 20$, *Unicast* and *Paint-Dense* generate $316{,}158$ and $3{,}501$ bytes, resp. Figure 5.21(b) shows that by taking into account $\mathcal{S}$, *Paint-Sparse* incurs even lower outgoing traffic than *Paint-Dense*; the gap is wider with fewer (sparser) subscriptions. Figure 5.21(c) shows that approximation further relieves any potential message injection bottleneck at the server.

Figure 5.22(a) provides more details on the outgoing traffic produced by different approaches. Although outgoing traffic increases for all approaches as k increases, the approaches presented in this chapter clearly outperform *Unicast*. *Paint-Dense* generates 1.5 orders of magnitude less outgoing traffic than *Unicast*, whereas *Paint-Sparse*, *Paint-Dense*(1), and *Paint-Sparse*(1) generate between 2 and 2.5 orders of magnitude less. Since the number of messages generated by *Paint-Dense* and *Paint-Dense*(1) is invariant to m, their lead over *Unicast* can widen arbitrarily as subscription density increases. The same trend holds for *Paint-Sparse* and *Paint-Sparse*(1); they always produce no more messages than *Paint-Dense* and *Paint-Dense*(1), resp.

For approximation algorithms, Figure 5.22(b) shows that increasing ε effectively decreases server outgoing traffic.

Figures above only show average outgoing traffic. When we look at the maximum amount of outgoing traffic from the server per event (which reveals bottlenecks better than the average) in Figure 5.23(a), we see an even bigger (multiple orders of magnitude) advantage of the presented approaches over *Unicast*. For *Unicast*, the maximum ongoing traffic is proportional to m, but remains the same when k varies because the number of affected subscriptions does not depend on k in the worst case (e.g., when the most popular object's y-value is dramatically changed). When



(a) Various approaches (b) Varying ε for *Paint-Dense*(ε)

FIGURE 5.22: Average outgoing traffic (# bytes) from server per event.

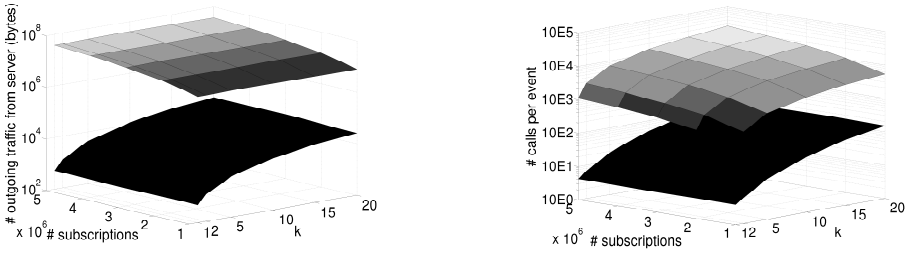
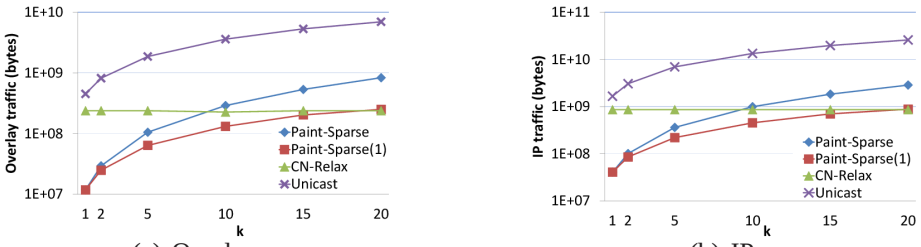


FIGURE 5.23: *Unicast* (top) vs. *Paint-Sparse* (bottom). (a) Maximum outgoing traffic from server for an event. (b) Number of calls per event.

$m = 5{,}000{,}000$, *Unicast*'s maximum outgoing traffic is $39{,}998{,}000$ bytes, compared with only $31{,}752$ bytes for *Paint-Sparse* (with $k = 20$).

Traffic in broker network. Figures 5.24(a) and 5.24(b) show the amounts of overlay and IP traffic (resp.) incurred per event in the broker network, averaged over all events in the workload. Trends in these two figures are consistent. *Unicast* performs worst among all approaches for all values of k tested and that *Paint-Sparse* leads *Unicast* by an order of magnitude. Furthermore, approximation is effective for reducing in-network traffic, as evidenced by *Paint-Sparse*(1). *CN-Relax* generates the same amount of in-network traffic for all k because it ignores ranking. While it may appear here that *CN-Relax* is attractive when $k > 10$ (largely because *CN-Relax* needs not be concerned with exposed objects), bear in mind that 1) *CN-Relax* requires subscriptions to maintain all objects within their ranges, which is expensive; and 2) *CN-Relax* generates excessive last-hop traffic, as we will see next.

Redundancy in messages received by subscriptions. Table 5.1 shows the total number of messages received by subscriptions per event (averaged over the workload) for *Paint-Sparse* (or any exact algorithm). Table 5.2 shows the overall redundancy in messages received by subscriptions



(a) Overlay (b) IP

FIGURE 5.24: Traffic in broker network per event.

Table 5.1: Total number of messages received by subscriptions per event.

k	1	2	5	10	15	20
Paint-Sparse	444.141	939.76	2211.14	4190.21	6082.94	7904.26

(averaged over the workload) for the approximate approaches. Note that all exact approaches would have 0 redundancy, and an approximate approach would effectively be exact if $1/\varepsilon \geq k$. Clearly, *CN-Relax* sends a lot of unnecessary messages to subscriptions, negating the advantages in its serverless approach and its relatively lower broker network traffic when $k > 10$. For the approximate approaches, as ε increases, their reduction in traffic from the server and within the broker network comes at the expense of higher redundancy. Still, they offer a spectrum of user-controllable trade-offs that are more attractive than the two extremes: exact algorithms on one hand and *CN-Relax* on the other.

Server processing cost. Figure 5.23(b) gives a high-level view of the average number of calls per event to the underlying data structures made by *Unicast* and *Paint-Sparse*. Tables 5.3 (varying k) and 5.4 (varying m, the number of subscriptions) offer a more detailed breakdown and comparison. As k or m increases, both *Paint-Sparse* and *Unicast* make more calls, but *Unicast* makes orders of magnitude more than *Paint-Sparse*.

Table 5.4 shows that the number of $\min_y$ calls by *Unicast* is linear in m and *Paint-Dense* is invariant to m. For *Paint-Sparse*, when m increases, there are fewer inessential exposed objects, so *Paint-Sparse* needs to examine more exposed objects during a rank-lowering update. However, our experiments show that the number of calls is increased only by a factor of roughly 2 even with dense subscriptions; therefore, our approach is much more scalable.

Table 5.2: Redundancy in messages received by subscriptions.

Approaches	$k = 1$	2	5	10	15	20
Paint-Sparse$(.125)$	0	0	0	0.015	0.0234	0.034
Paint-Sparse$(.25)$	0	0	0.027	0.061	0.070	0.080
Paint-Sparse$(.5)$	0	0	0.11	0.14	0.16	0.17
Paint-Sparse(1)	0	0.16	0.29	0.30	0.32	0.34
CN-Relax	1440.2	680.14	288.49	151.76	104.23	79.98

Table 5.3: Average number of calls per event; increasing k.

k	*Paint-** # first_k	*Paint-Dense* # $\min_y$	*Paint-Sparse* # $\min_y$	*Paint-Sparse* # snap	*Unicast* # $\min_y$
1	2	1.12	0.72744	1.2284	444.141
2	2	1.73	1.08988	2.95254	1086.44
5	2	3.57	2.59596	12.06538	2846.1
10	2	6.60	5.5568	40.69192	5453.64
15	2	9.63	8.57692	84.65234	8032.39
20	2	12.63	11.58048	143.20416	10587.4

Table 5.4: Average number of calls per event; increasing m.

m ($\times 10^5$)	# $\min_y$	# first_k	# snap
2	3.52	2	29.87
8	4.84	2	37.34
40	5.94188	2	42.0509
100	6.26988	2	42.86982
Dense	6.60	2	43.36

Paint-Sparse

m ($\times 10^5$)	# $\min_y$
2	545.23
8	2181.49
40	10906.4
100	27267.5

Unicast; $k = 10$

Batch processing. Next, the effectiveness of the batch processing algorithm, *Paint-Batch*, is evaluated by comparing it with *Online*, which simply processes the batched event sequence one event at a time, and *Coalesce*, which coalesces events updating the same object into one before processing, but does not sort or group them into $\mathcal{E}^{\downarrow}$ and $\mathcal{E}^{\uparrow}$. The sequence contains $50{,}000$ events, and k varies from 1 to 20. Figure 5.26(a) compares the total number of messages generated over the sequence; Figure 5.26(b) compares the total number of messages received by all subscriptions; Figure 5.26(c) compares the total number of $\min_y$ calls. In all figures, *Paint-Batch*, with both coalescing and sorting optimizations, dominates the other approaches. The savings provided by sorting (between *Paint-Batch* and *Coalesce*, especially in the number of messages received by subscriptions) are significant, though they are dwarfed by the savings provided by coalescing.

Trends across synthetic workloads. Results for other workloads are similar, and exhibit trends that confirm intuition. Figure 5.25(a) shows the ratio between the number of messages generated

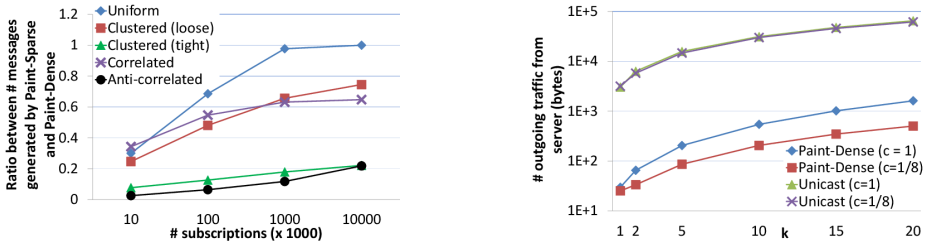
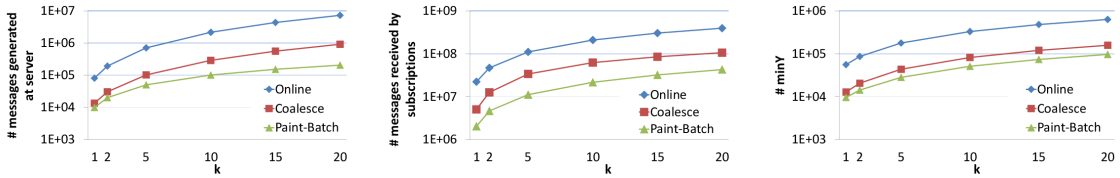


FIGURE 5.25: (a) *Paint-Sparse* vs. *Paint-Dense* for various workloads, with # objects = 1,000. (b) Average outgoing traffic from server per event, with y-value changes following a Gaussian distribution.



(a) Total number of messages generated

(b) Total number of messages received by subscriptions

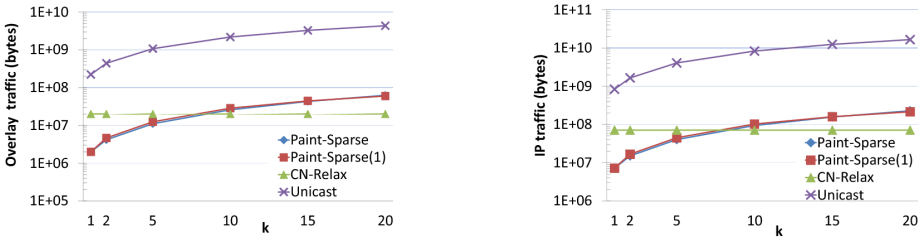
(c) Total number of $\min_y$ calls

FIGURE 5.26: Batch processing approaches.

by *Paint-Sparse* and *Paint-Dense* for various workloads. With knowledge of $\mathbb{S}$, *Paint-Sparse* (as well as *Paint-Sparse*(ϵ), which is not shown here) generates less traffic with more clustered subscriptions, because of more opportunities for skipping empty regions in $\mathbb{S}$. The ratio is 1 with ten million uniformly distributed subscriptions, which are basically dense. Furthermore, *Paint-Sparse* skips a greater number of inessential exposed objects for the anti-correlated workload than for the correlated one.

In practice, y-values of objects rarely change in a completely random fashion. To see how this observation impacts the performance of the presented algorithms, instead of choosing new y-values uniformly at random, the difference between the new and old y-values follows a Gaussian distribution with standard deviation set to $c/8$ times the length of the range of possible y-values. A smaller c means changes are less volatile. Figure 5.25(b) shows the traffic from the server for two settings of c. It is evident that *Paint-Dense* generates fewer messages when c is smaller because fewer objects are exposed by less volatile value (and hence rank) changes. The traffic under Unicast is approximately the same for both $c = 1$ and $c = 1/8$.

Yahoo! Finance data. Results for Yahoo! Finance workload are largely consistent with other results presented in this section, so some samples are shown here comparing *Paint-Sparse*, *Paint-*



(a) Overlay (b) IP

FIGURE 5.27: Traffic in broker network per event; Yahoo! workload.

Table 5.5: Redundancy in messages received; Yahoo! workload.

Approaches	$k = 1$	2	5	10	15	20
Paint-Sparse(1)	0	0.19	0.25	0.35	0.39	0.38
CN-Relax	706.51	361.22	147.13	72.67	48.38	36.11

Table 5.6: Average number of calls per event; Yahoo! workload.

	k	# $\min_y$	# first_k	# snap
Paint-Sparse	1	0.5	2	1
	2	0.5	2	2
	5	0.51	2	5.05
	10	0.54	2	10.35
	15	0.57	2	16.02
	20	0.61	2	22.18

	k	# $\min_y$
Unicast	1	176.01
	2	409.44
	5	1050.48
	10	2278.27
	15	3510.16
	20	4614.24

Sparse(1), *CN-Relax*, and *Unicast*. In terms of outgoing traffic from the server, this workload allows *Paint-Sparse* and *Paint-Sparse*(1) to inject a significantly fewer number of messages into CN than other workloads, because the y-values (price-to-earning ratios) only change slightly for most events; consequently, most rank-lowering updates expose only a few objects. In terms of traffic in the broker network, Figures 5.27(a) and 5.27(b) show that *Paint-Sparse* and *Paint-Sparse*(1) generate two orders of magnitude less traffic than *Unicast*. While *CN-Relax* again seems attractive around $k = 10$, it does poorly with the next metric, redundancy in messages received by subscriptions, shown in Table 5.5. Here, *CN-Relax* results in far more unnecessary traffic to subscriptions with double- and triple-digit redundancy, compared with less than 0.4 for *Paint-Sparse*(1) (and 0 for *Paint-Sparse* because it is exact). Finally, in terms of server processing cost, Table 5.6 shows that *Paint-Sparse* makes few calls. On the other hand, the number of $\min_y$ calls remains huge for *Unicast*, because it still checks all subscriptions in $(\ell, x_i] \times [x_i, r) \subseteq \mathbb{S}$ even though the majority of events affect no subscriptions.

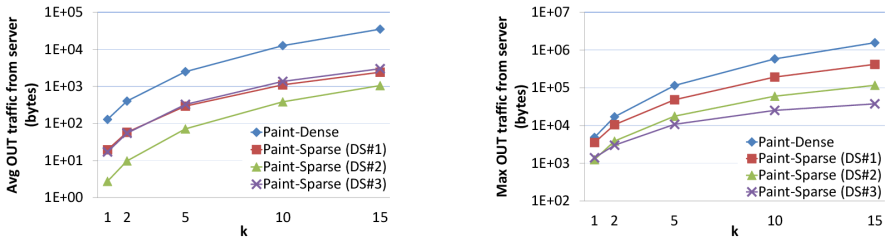


FIGURE 5.28: (a) Average outgoing traffic from server per event; (b) Maximum outgoing traffic from server per event.

Table 5.7: Number of $\min$ queries.

Approaches	$k = 1$	2	5	10	15
Paint-Dense	6.35	11.31	26.13	50.48	74.40
Paint-Sparse Dataset#1	5.68	9.88	23.07	45.85	68.826
Paint-Sparse Dataset#2	5.40	8.97	19.45	36.97	54.67
Paint-Sparse Dataset#3	5.51	9.36	21.12	41.13	61.07

5.7.2 1.5*-dimensional range subscriptions*

There are $10{,}000$ objects, whose $x^{(1)}$-values and $x^{(2)}$-values are uniformly distributed over the possible $x^{(1)}$-value and $x^{(2)}$-value ranges. The number of subscriptions is $400{,}000$ subscriptions. The following subscription distributions are considered:

- *Dataset #1:* Uniformly pick two random numbers in the $x^{(1)}$-value range. $\ell^{(1)}$ and $r^{(1)}$ are set to be the smaller and larger ones, respectively. $r^{(2)}$ is uniformly chosen in the $x^{(2)}$-value range.

- *Dataset #2:* $\ell^{(1)}$ is uniformly chosen in the $x^{(1)}$-value range. $r^{(1)} - \ell^{(1)}$ is set to be the minimum width that covers $1{,}000$ objects. $r^{(2)}$ is uniformly chosen in the $x^{(2)}$-value range.

- *Dataset #3:* Same as Dataset #2, except that $r^{(1)} - \ell^{(1)}$ is set to be the minimum width that covers 100 objects.

Figures 5.28(a) and 5.28(b) shows the average and maximum outgoing traffic (in bytes) from the server per event update. Tables 5.7 and 5.8 show the number of calls per event.

Table 5.8: Number of snap queries.

Approaches	$k = 1$	2	5	10	15
Paint-Sparse Dataset#1	4.55	13.86	88.43	473.43	1361.90
Paint-Sparse Dataset#2	4.08	10.92	51.44	213.40	561.45
Paint-Sparse Dataset#3	4.29	12.28	70.69	360.45	1022.86

5.8 Related Work

Much work on scalable processing and notification of subscriptions has been done in the context of publish/subscribe systems (e.g., [23, 34, 93]), but traditionally they consider only selection queries over message attributes. Recent work seek to extend them to support more complex subscriptions (e.g., [53, 41, 40, 42]), or use them for scalable implementation of distributed stream processing [124] and query result caching [59]. The work most relevant to this chapter is [41], which discusses scalable processing and dissemination of range top-1 subscriptions. This chapter builds on their approach of leveraging CN for efficient dissemination. However, as demonstrated in this chapter, the case of $k > 1$ is considerably more complex and requires new algorithms data structures; this chapter also considers batch updates and approximate solutions.

Other recent work on publish/subscribe has also addressed ranking, but with various different subscription semantics; little is known about how best to support standard range top-k subscriptions. Drosou et al. [55] consider ranking events by relevance and diversity. Machanavajjhala et al. [79] consider the reverse problem—finding most relevant subscriptions for a published event. In the sliding window model, Pripuzic et al. [96] maintains a buffer to store relevant events that have a high probability of entering a top-k result in the future, and Haghani et al. [63] continuously monitor top-k queries over incomplete data streams. Lu et al. [78] consider an approximate top-k real-time publish/subscribe model, in which each subscriber approximately receives the k most relevant publications before a deadline.

Range top-k querying is well studied in the database literature, both in terms of access method design (e.g., [111]), and integration with relational query processing and optimization (e.g., [77]). The key difference is that this chapter focuses on a different dimension of scalability here: instead of making a single range top-k query scale over a large dataset, this chapter considers how to scale over a large number of ongoing range top-k queries.

This chapter is related to incremental maintenance of materialized top-k views. [120] handles the challenge that an object "escaping" from the top k requires obtaining the new k-th ranked

object. The idea is to reduce the expected amortized maintenance cost over time by maintaining a top-k' view where $k' \geq k$ is allowed to vary. This approach (which optimizes across time) complements ours (which optimizes across subscriptions), and will be interesting to explore in conjunction with the approximate algorithms.

The problem studied in this chapter is related to that of *reverse top-k queries* [115], where, given a data update, affected queries are identified and their results are updated. Their definition of top k is different from ours, however: queries do not specify range conditions but instead vectors of weights that customize relative importance of different ranking criteria. Also, the issue of efficiently notifying affected queries over a network is not considered.

There also has been much research on top-k processing in a distributed setting, e.g., [20], [33], [81], [87]. Most previous work focuses on computing or monitoring the result for a single top-k query over a set of distributed sources, where each source provides either individual object scores or partial scores that must be aggregated across sources before being used for ranking. Processing can be pushed inside the network to reduce communication, e.g., [80, 107]. Compared with the work above, the problem setting of this chapter is inverted—instead of having one query over many distributed objects, there are many distributed subscriptions over one stream of object updates, which call for different techniques. Nonetheless, some ideas from distributed top-k monitoring [20, 107]) may be interesting to explore as future work. Namely, some solutions for distributed top-k monitoring involve installing conditions at the sources that trigger reporting; intuitively, lowly-ranked objects with little chance of entering the top k are associated with loose reporting conditions with reduced monitoring costs. The question of applying this approach to the setting of this chapter, however, is whether a large number of reporting conditions (mn) can be handled.

5.9 Conclusion

This chapter has tackled the problem of supporting a large number of range top-k subscriptions in a wide-area network. The dual challenges of subscription processing and notification dissemination are addressed by carefully separating and interfacing these tasks in a way that achieves efficiency with off-the-shelf dissemination networks and without increasing system complexity. The techniques presented in this chapter are based on a geometric framework, enabling us to characterize the subset of subscriptions affected by an event as a region in an appropriately defined space, and solve the problem of notifying affected subscriptions as one of tiling the region with basic shapes.

The array of techniques that have been developed—ranging from those that use the knowledge of subscriptions to those that do not, from event-at-time to batch processing, from exact to approximate, and from one-dimensional to multi-dimensional ranges—speak to the power of this framework. Theoretical analysis and empirical evaluation show that the presented approach holds substantial advantages over less sophisticated ones.

As mentioned in Section 5.1, the techniques presented in this chapter can be applied to other application settings. In essence, this chapter has devised an effective way to divide the problem of supporting a large number of stateful subscriptions into two tasks: one that computes a compact description of the changes, and one that further uses this description to update affected subscriptions. The first task is shielded from the complexity of handling subscriptions, while the second is shielded from the complexity of handling objects. This division allows each task to be scaled up independently. This chapter uses CN to scale up dissemination for the second task, but there are more possibilities. 1) In settings where result updates do not need to be delivered over a network, the second task of updating subscriptions can be scaled up in an embarrassingly parallel fashion, without duplicating the effort of the first task or requiring each processing node to maintain the set of objects. 2) Instead of using a single server to perform the first task, the database of objects can be distributed across multiple nodes, which process incoming events and generate outgoing messages in a distributed fashion. Details are available in Section 5.6.3. This extension allows us to handle the general publish/subscribe setting where events originate from multiple, distributed publishers.

6

Dissemination Network Design

This chapter studies the problem of assigning subscribers to brokers in a wide-area content-based publish/subscribe system. A good assignment should consider both subscriber interests in the event space and subscriber locations in the network space, and balance multiple performance criteria including bandwidth, delay, and load balance. The resulting optimization problem is NP-complete, so systems have turned to heuristics and/or simpler algorithms that ignore some performance criteria. Evaluating these approaches has been challenging because optimal solutions remain elusive for realistic problem sizes. In this chapter, a Monte Carlo approximation algorithm with good theoretical properties and robustness to workload variations is developed to enable proper evaluation. The algorithm combines the ideas of linear programming, randomized rounding, coreset, and iterative reweighted sampling to make the problem computationally feasible. Because of its theoretical properties and robustness to workload variations, it can serve as a reasonable yardstick to evaluate other algorithms. In the evaluation section, we will see that with its help, a simple greedy algorithm works well for a number of workloads, including one generated from publicly available statistics on Google Groups. The hope is that the presented algorithms are not only useful in their own right, but the presented principled approach toward evaluation will also be useful in future evaluation of solutions to similar problems in content-based publish/subscribe.

6.1 Introduction

A wide-area publish/subscribe system typically consists of an overlay network of *brokers*. *Events* originate from *publishers*, and are delivered by the brokers to interested *subscribers*. Traditional publish/subscribe is *topic-based*, where subscribers subscribe to a set of predefined topics such as "Apple news" or "American Idol." *Content-based* publish/subscribe, on the other hand, allows a subscriber to express an interest as a Boolean predicate against values of attributes inside events. For example, a subscriber may subscribe to eBay antique auctions with seller rating higher than 90% and starting bid between $100 and $200. Only events matching the predicate will be delivered to the subscriber. Content-based publish/subscribe is of interest to both database and networking communities [13, 53, 93, 98], because it must address the dual challenges of subscription matching in an event space and event dissemination in the network space.

An important problem in content-based publish/subscribe is *subscriber assignment*. Each subscriber needs to be assigned a broker responsible for forwarding matching events to this subscriber. Intuitively, we would like to assign subscribers with similar interests to the same broker, so that an event delivered to the broker could serve many subscribers. If all subscribers assigned to the broker have similar interests, only a subset of all possible events needs to go through the broker. At the same time, we may not want to assign a subscriber to a broker located far away in the network, because doing so increases delivery latency and communication cost. Finally, we should not assign too many subscribers to one broker because it could create a performance bottleneck and delays event delivery. Balancing these considerations—similarity of interests in the event space, proximity of locations in the network space, and balance of load across brokers—is a difficult optimization problem.

The Need for a Yardstick. There is a good amount of previous work on subscriber assignment and related problems; see Section 6.7 for details. Most approaches ignore some aspects of the problem or employ heuristic algorithms. For example, Aguilera et al. [13] assign subscribers to their closest brokers in the network, ignoring subscriber interests. On the other hand, Diao et al. [53]

make assignment based on similarity of interests, without considering network latency. Papaemmanouil et al. [91] present a general optimization framework that considers multiple performance criteria, but relies on an iterative method to explore the solution space through local adjustments of dissemination trees.

It is understandable and often necessary to employ heuristics for subscriber assignment, because the problem in general is NP-complete. Evaluating these heuristics, however, is frustratingly difficult. How close are their solutions to the optimal? How well do they work on large, realistic workloads? Because of the problem's inherent complexity, optimal solutions for realistic problem sizes are computationally elusive and often unavailable for comparison. What would be a good yardstick then? Could yardsticks be solutions to simpler problems that ignore some performance constraints, since they are easier to compute and can act as lower bounds for the optimal solution?

Contributions. A main goal of this chapter is to find a better yardstick for evaluating the performance of various algorithms for the subscriber assignment problem. An algorithm called *SLP*, a shorthand for ***S**ubscriber Assignment by **L**inear **P**rogramming*, is proposed in this chapter. SLP jointly considers both subscriber interests in the event space and subscriber locations in the network space, and balances multiple performance criteria including bandwidth, delay, and load balance. While SLP's solution is not guaranteed to be optimal, it has provable properties that make it robust to workload variations, and reasonable as a yardstick for evaluating other algorithms. Moreover, a by-product of running SLP (the LP fractional solution) gives us another useful indicator of how close a solution is to the optimal.

This chapter also presents *Gr**, a simple offline greedy algorithm for subscriber assignment that presorts the subscribers in a particular way before assigning them one by one. SLP is used as a yardstick to evaluate Gr* and a number of other algorithms. With the help of SLP, this chapter is able to conclude, with confidence, that Gr* works very well for most (but not all) of the workloads tested. The evaluation also reveals that simpler algorithms that ignore one performance criterion or another are poor yardsticks, because their solution cannot offer meaningful bounds on what can be realistically achieved when considering all constraints.

Another major obstacle for evaluation is the lack of publicly available, realistic workloads for content-based publish/subscribe. Information about subscribers (interests and locations) is rarely disclosed because of privacy concerns and commercial interests. Lack of widely deployed systems with powerful subscription languages also contributes to the difficulty. Thus, researchers have often resorted to synthesized workloads. However, simplistic workload generators run the risk of missing interesting patterns of clustering and overlap among subscriber interests, and correlations between subscriber interests and locations, which may influence the evaluation of subscriber assignment algorithms. Therefore, beyond simple synthetic workloads used for evaluation by previous work, The algorithms are also evaluated using workloads generated from publicly available statistics on Google Groups [121], which is believed to be closer to (at least one) reality.

SLP is computationally feasible on realistic problem sizes; it has been run on workloads consisting of hundreds of brokers and a million subscribers. SLP is made scalable by combining a suite of techniques, including randomized rounding, coreset, and iterative reweighted sampling. While SLP is slower than the simpler algorithms, its solution quality makes it well worthwhile in some settings, such as initial subscriber assignment, periodical re-optimization, and especially comparison with and evaluation of other algorithms.

6.2 Problem Statement

Let $\mathbb{N}$ denote the *network space*. For simplicity, $\mathbb{N}$ is assumed to be a multi-dimensional Euclidean space, obtained by standard Internet embedding techniques [48, 76, 88]; Euclidean distance between two points approximates the network latency between them. Let $P \in \mathbb{N}$ be the *publisher* and $\mathcal{S} = \{S_1, \cdots, S_m\} \subseteq \mathbb{N}$ be a set of m *subscribers*.

P publishes *events*, each of which is represented as a point in the *event space* $\mathbb{E}$. $\mathbb{E}$ is assumed to be the d-dimensional Euclidean space $\mathbb{R}^d$. Each subscriber S_i has an *interest* s_i, which is assumed to be a d-dimensional rectangle in $\mathbb{E}$.[1] S_i receives an event $e \in \mathbb{E}$ if $e \in s_i$.

Events are disseminated to subscribers using a set $\mathcal{B} = \{B_1, \cdots, B_n\} \subseteq \mathbb{N}$ of n *brokers*. P

[1] Without loss of generality, each subscriber is assumed to have one interest; an individual with multiple interests can be modeled as multiple subscribers located at the same point in $\mathbb{N}$.

and $\mathcal{B}$ form a *dissemination network*, which is assumed to be a tree $\mathcal{T}$ rooted at P. A leaf of $\mathcal{T}$ is called a *leaf broker*. A *subscriber assignment* $\Sigma : \mathcal{S} \to \mathcal{B}^{\text{Leaves}}$ connects each subscriber to a leaf broker.

Filters. Each broker B_i is associated with a *filter* $f_i \subseteq \mathbb{E}$ such that if a broker B_j (resp. subscriber S_j) is a descendant of B_i, then $f_j \subseteq f_i$ (resp. $s_j \subseteq f_i$). This condition is referred to as the *nesting condition*. An event e is passed to a broker B_i if $e \in f_i$. To ensure simplicity and efficiency in implementing this forwarding logic, f_i is required to be the union of at most α_i rectangles, for a user-defined small constant α_i which is called *filter complexity*. That is, $f_i = \bigcup_{R \in F_i} R$, where F_i is a set of rectangles in $\mathbb{E}$ and $|F_i| \leq \alpha_i$. In the special case of $\alpha_i = 1$ for all brokers B_i, $\mathcal{T} \cup \Sigma$ becomes a bounding box hierarchy like an R-tree. However, α_i is allowed to be 1. Figure 6.1 shows an example of f_i for $\alpha = 1$ and 2; the red points (events) are the false positive since they do not hit the filters of B_i's children—B_1, B_2, B_3, and B_4.

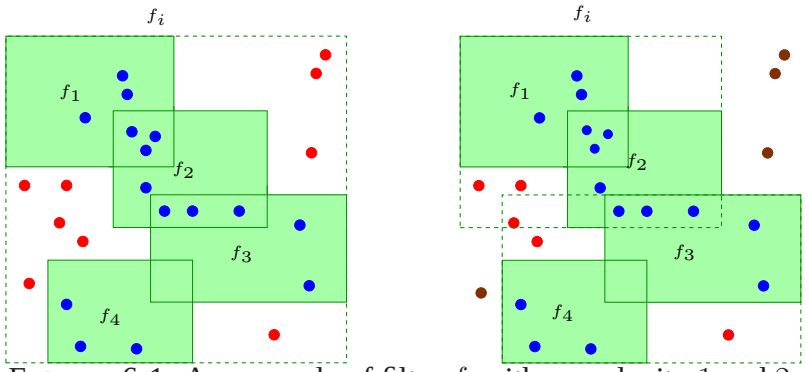


FIGURE 6.1: An example of filter f_i with complexity 1 and 2.

Bandwidth. We are interested in minimizing $Q(\mathcal{T})$, the *expected total bandwidth consumption* (or *bandwidth* for short) of $\mathcal{T}$. $Q(\mathcal{T}) = \sum_{B_i \in \mathcal{B}} Q(B_i)$, where $Q(B_i)$ is the expected bandwidth *into* broker B_i. The bandwidth required for leaf brokers to deliver events to subscribers is ignored because the total does not depend on the subscriber assignment. If events are uniformly distributed, $Q(B_i)$ is defined as the volume of f_i, $\text{Vol}(f_i)$. Our approach can be extended to a non-uniform event distribution π, in which case $Q(B_i) = \int_{f_i} \pi(e)de$.

Choosing $\alpha_i > 1$ can reduce bandwidth into a broker, as multiple rectangles summarize child filters or subscriber interests more precisely than a single rectangle, but at the cost of increasing storage and processing overhead at the broker.

Latency. We want to bound the latency of delivering events to each subscriber S_j. A natural requirement is made in this chapter: for a subscriber assignment Σ to be valid, the network latency of the path in $\mathcal{T} \cup \Sigma$ from the publisher to each subscriber S_j must not exceed the user-defined *maximum allowable latency* δ_j for S_j. Here, the path latency is the sum of distances in $\mathbb{N}$ between consecutive points on the path.

The approach to be presented in this chapter can be extended to handle other form of latency constraints, such as one that bounds only the last-hop latency to each subscriber (from the broker it is assigned to). More sophisticated constraints that account for broker processing delays can be enforced by additionally imposing load balance constraints described below.

Load Balance. We also want to ensure that not too many subscribers are assigned to one leaf broker, otherwise, the processing cost of a broker (matching incoming events against subscribers and notifying the interested subscribers) would become too expensive. Without loss of generality, assume that $B_1, \cdots, B_l$ are the l leaf brokers in $\mathcal{B}$. Each leaf broker B_i is associated with a user-defined *capacity fraction* $\kappa_i \in [0, 1]$, such that $\sum_{i=1}^{l} \kappa_i = 1$. Perfect load balance happens when each B_i is assigned $\kappa_i m$ subscribers, but it is unnecessary and often undesirable as it may sacrifice other performance measures. Let m_i be the number of subscribers assigned to leaf broker B_i; the *load balance factor* (*lbf*) of the assignment be defined as $\max_{1 \le i \le l} \frac{m_i}{\kappa_i m}$. The user is allowed to cap the lbf at $\beta_{\max}$ and specify a *desired lbf* $\overline{\beta}$, where $\beta_{\max} > \overline{\beta} > 1$. We try to find an assignment with lbf within $\overline{\beta}$; failing that, we try to find an assignment with lbf within $\beta_{\max}$ and as close to $\overline{\beta}$ as possible. The pair $(\overline{\beta}, \beta_{\max})$ allows the user to encourage load balance towards the desired level without rewarding assignments that "over-balance."

The Problem. The *subscriber assignment problem* (*SA*) is defined as follows: Given P, $\mathcal{B}$, $\mathcal{B}^{\text{Leaves}} \subseteq \mathcal{B}$, $\mathcal{S}$, $\mathcal{T}$, filter complexities $\boldsymbol{\alpha} = \{\alpha_1, \ldots, \alpha_m\}$, maximum allowable latencies $\boldsymbol{\delta} = \{\delta_1, \ldots, \delta_m\}$, leaf broker capacity fractions $\boldsymbol{\kappa} = \{\kappa_1, \ldots, \kappa_l\}$, as well as parameters $\overline{\beta}$ and $\beta_{\max}$, compute an assignment $\Sigma : \mathcal{S} \rightarrow \mathcal{B}^{\text{Leaves}}$ and filters for all brokers, such that the filter nesting condition and complexity constraint are satisfied by all filters, the latency constraint is satisfied at each subscriber, and the load balance factor is no more than $\overline{\beta}$ (or as close to $\overline{\beta}$ as possible and no more than $\beta_{\max}$). The assignment with the minimum expected total bandwidth $Q(\mathcal{T})$ will be returned. By reducing the standard set cover problem [114] to SA, it can be shown that SA is NP-complete.

Theorem 26. *The decision version of the broker-subscriber assignment problem is NP-complete.*

Proof. First, the geometric set cover problem, which is well-known to be NP-complete, can be reduced to the subscriber assignment problem. The geometric set cover decision problem is formulated as follows: Given a set $\mathcal{S}$ of m points in $\mathbb{R}^2$, a set $\mathcal{B}$ of n points in $\mathbb{R}^2$, and an integer k, does there exist a set $\mathcal{B}' \subseteq \mathcal{B}$ of size k, such that $\max_{S \in \mathcal{S}} \min_{B \in \mathcal{B}'} \parallel S - B \parallel \leq 1$?

Let $\mathcal{S}$ be the subscriber set and $\mathcal{B}$ be the leaf broker set. Subscriber interest s_i is set to $[0, 1]^2$ for $1 \leq i \leq m$, and desired lbf $\overline{\beta}$ is set to n. The dissemination tree $\mathcal{T}$ is constructed as follows: Choose publisher P to be the centroid of $\mathcal{B}$ in $\mathbb{R}^2$. Connect each leaf broker $B \in \mathcal{B}$ to P by a separate path consisting of 2 edges, such that the total length of the path is $\beta \geq \max_j \parallel P - B_j \parallel$. It can be checked that there exists a set cover of size k iff there is a valid subscriber assignment of bandwidth $Q(\mathcal{T}) \leq 2k$. Since the subscriber assignment problem generalizes the geometric set cover problem, it is NP-hard. Finally, the subscriber assignment problem is in NP because the resulting filter and subscriber assignments can be verified in polynomial time. □

An Example of SA. Refer to Figure 6.2. Both the event space and network space, shown as the horizontal and vertical axes (resp.), are one-dimensional in this simple example. The horizontal thin red line segments represent subscriber interests. The horizontal thick green lines represent filters. The filter complexities for brokers B_1, B_2, and B_3 are 2, 1, and 1, respectively. $\beta_{\max}$ is set to 1.5, so at most three subscribers can be assigned to each broker. The arcs (with arrows)

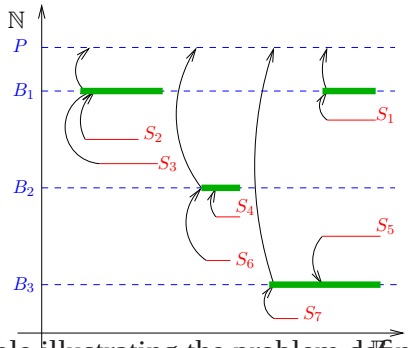


FIGURE 6.2: An example illustrating the problem definition in low dimensions.

indicate the assignment of subscribers to brokers, as well as the connection from brokers to the publisher. Although assigning subscriber S_5 to broker B_1 can further reduce bandwidth, B_1 will become overloaded. Assigning S_1 to B_3 can also reduce bandwidth, but the latency constraint for S_1 will be violated.

6.3 Two Greedy Algorithms

This section presents two simple greedy algorithms for SA, both aimed at minimizing bandwidth while meeting the latency and the load-balance constraints.

Online Greedy (Gr) This algorithm assigns subscribers sequentially to leaf brokers. It need not know the set of subscribers from the start. It considers the effect of incorporating the new subscriber into existing filters in the event space, in a way similar to R-tree splitting heuristics. For each subscriber $S_j \in \mathcal{S}$, the *cost* of assigning S_j to a leaf broker B_i is defined to be the sum of least volume enlargement of filters over the path in $\mathcal{T}$ from the publisher to B_i, such that the nesting condition is preserved. More specifically, let $f_i = \bigcup_{R \in F_i} R$ be the current filter of broker B_i. If S_j is assigned to B_i, one of the rectangles in F_i needs to be enlarged to contain subscriber interest s_j. The least volume enlargement of f_i can be computed by finding the rectangle whose expansion results in the least increment of the expected bandwidth $Q(B_i)$. Gr identifies a set of *candidate brokers* (defined below) for S_j, and then greedily assigns S_j to the candidate broker with the minimum cost. It breaks a tie by choosing the least loaded broker (i.e., one with the minimum $\frac{m_i}{\kappa_i |\mathcal{S}|}$, where m_i is the number of subscribers already assigned to it).

B_i is a *candidate broker* for S_j if the following conditions are met: 1) Assigning S_j to B_i satisfies the user-defined latency constraint; 2) B_j will not be overloaded by this assignment; i.e., $\frac{m_i+1}{\kappa_i|\mathcal{S}|}$ is no more than a user-specified lbf. (This lbf can be set initially to $\overline{\beta}$; it can be increased if no feasible solution is found, eventually to $\beta_{\max}$.)

Offline Greedy (Gr*) This algorithm is an offline and more expensive variant of Gr. Each subscriber is processed in the exact same way as Gr. However, Gr* first sorts and then processes the set of subscribers in ascending order of the cardinality of their candidate broker sets. Intuitively, by deferring the processing of subscribers with more choices, it reduces the chance that Gr* will be forced into a costly decision due to lack of choices. Note that the assignment of earlier subscribers may restrict the choices available to later subscribers; hence, Gr* updates the ordering of remaining subscribers whenever a broker becomes fully loaded. As we will see in Section 6.6, Gr* not only consumes lower bandwidth than Gr but also produces more balanced loads than Gr.

6.4 One-Level SA

We now turn to a more sophisticated algorithm, SLP. This section describes SLP$_1$, an algorithm for solving the one-level version of SA, in which all brokers are directly connected to the publisher in $\mathcal{T}$. Section 6.5 extends the solution to a multi-level $\mathcal{T}$. For a better flow of the chapter, all proofs in this section are presented at the end of the chapter.

Although SA can be written as an integer programming problem, solving it directly is computationally intractable even for the one-level version. Realistic workloads involving hundreds of thousands of subscribers easily overwhelm the most sophisticated solvers. To tame complexity, a carefully simplified problem is first solved to obtain a preliminary, but nonetheless good, assignment of filters to brokers; it is then used to derive the final solution to the full problem. The three-step strategy, illustrated in Figure 6.3, is as follows.

1. *Preliminary filter assignment.* The heart of SLP$_1$, this step produces a preliminary filter assignment $\Phi = \{\varphi_1, \ldots, \varphi_m\}$ where broker B_i is assigned filter φ_i. This step considers all factors simultaneously in optimization—bandwidth, latency, and load balance—using LP

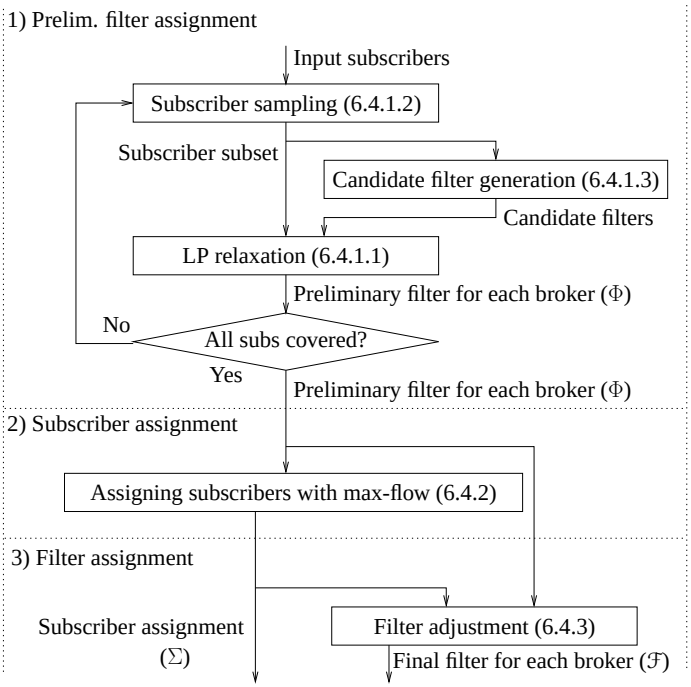


FIGURE 6.3: Overview of SLP_1.

relaxation and randomized rounding. To keep the LP size manageable, instead of solving LP on all subscribers and all possible filters, LP is iteratively run on small-size representative sets (coresets) of subscribers and candidate filters.

2. *Subscriber assignment.* Given a preliminary filter assignment Φ, this step considers the full set of subscribers and computes the subscriber assignment $\Sigma : \mathcal{S} \to \mathcal{B}^{\text{Leaves}}$. Since the filters are already given, this step focuses on load balancing while meeting latency constraints, using a max-flow algorithm.

3. *Filter adjustment.* Given Φ and Σ, this step further refines the filters and enforces the maximum filter complexity. Let $\mathcal{F} = \{f_1, \cdots, f_n\}$ be the resulting set of filters. The algorithm returns Σ and $\mathcal{F}$.

6.4.1 Preliminary Filter Assignment

This section presents the first step of SLP_1, $\mathsf{FilterAssign}(\mathcal{B}^{\text{Leaves}}, \mathcal{S})$ (Algorithm 9). Section 6.4.1.1 describes LPRelax, a subroutine for computing a filter assignment using LP relaxation. Calling this

Algorithm 9: Preliminary filter assignment algorithm.

```
1  FilterAssign(B^Leaves, S) begin
2      g ← 4;
3      while g ≤ |S| do
4          foreach S ∈ S do w(S) ← 1;
5          q ← 10g ln g;
6          for i ← 1 to 4g ln(|S|/g) do
7              repeat
8                  Q ← Random(S, w, q);
9                  Φ ← FilterAssignHelper(Q, B^Leaves, S);
10                 if Φ = ⊥ then return ⊥;
11                 if Violate((1 + ε)Φ, B^Leaves, S) = ∅ then
12                     return (1 + ε)Φ;
13                 V ← Violate(Φ, B^Leaves, S);
14             until Σ_{S∈V} w(S) ≤ (1/8) Σ_{S∈S} w(S) ;
15             foreach S ∈ V do w(S) ← 2w(S);
16         g ← 2g;
17     return ⊥;
18 end
19 FilterAssignHelper(Q, B^Leaves, S) begin
20     for j ← 0 to ln |S| do
21         S_b ← Random(S, 1, 10|B^Leaves|);
22         S_a ← Q ∪ S_b;
23         R ← FilterGen(S_a);
24         Φ ← LPRelax(B^Leaves, R, S_a, S_b);
25         if Φ ≠ ⊥ then return Φ;
26     return ⊥;
27 end
```

subroutine with all subscribers and all possible filters is impractical. Therefore, in Section 6.4.1.2, iterative reweighted sampling is used to obtain a coreset of subscribers to run LPRelax with. Section 6.4.1.3 presents a method for choosing a good subset of candidate filters to be considered by LPRelax.

6.4.1.1 LP Relaxation

First, the algorithm LPRelax$(\mathcal{B}^{\text{Leaves}}, \mathcal{R}, \mathcal{S}_a, \mathcal{S}_b)$ is described. It assigns each broker $B_i \in \mathcal{B}^{\text{Leaves}}$ a filter consisting of rectangles in $\mathbb{E}$ drawn from a given set $\mathcal{R} = \{R_1, \cdots, R_u\}$. $\mathcal{S}_a$ denotes the

subset of $\mathcal{S}$ considered by LPRelax; $\mathcal{S}_b \subseteq \mathcal{S}_a$ denotes the subset for which LPRelax enforces the load balance constraint (see (C3) below). Intuitively, we would like $\mathcal{S}_a = \mathcal{S}_b = \mathcal{S}$ and let $\mathcal{R}$ contain the minimum enclosing box of each non-empty subset of the subscriber interests, but doing so would make the algorithm quite expensive in practice. Therefore, a subset $\mathcal{S}_a \subseteq \mathcal{S}$ is carefully chosen so that a filter assignment with respect to $\mathcal{S}_a$ is also good with respect to the entire set $\mathcal{S}$, and choose a subset $\mathcal{S}_b \subseteq \mathcal{S}_a$ to facilitate load balancing. Later, Section 6.4.1.2 will address how to choose $\mathcal{S}_a$ and $\mathcal{S}_b$ (and why to distinguish them), and Section 6.4.1.3 will address how to choose $\mathcal{R}$.

For each subscriber $S_j \in \mathcal{S}_a$, let $\mathcal{B}_j \subseteq \mathcal{B}^{\text{Leaves}}$ be the subset of brokers that satisfy the user-defined latency constraint for S_j if S_j is assigned to them; let $\mathcal{R}_j = \{R_k \in \mathcal{R} \mid s_j \subseteq R_k\}$, i.e., the subset of given rectangles that contain S_j's interest.

SA is formulated as a mixed integer program. Two sets of Boolean variables x_{ij}, $y_{ik} \in \{0, 1\}$ are introduced for $i \in [1, n]$, $j \in \{j \mid S_j \in \mathcal{S}_a\}$, and $k \in [1, u]$, where $x_{ij} = 1$ iff subscriber S_j is assigned to broker B_i, and $y_{ik} = 1$ iff rectangle R_k is assigned to B_i as part of its filter.

Recall from Section 6.2 that we want to minimize $\sum_{B_i \in \mathcal{B}^{\text{Leaves}}} Q(B_i)$, but when $\alpha_i > 1$, using $Q(B_i) = \text{Vol}(f_i) = \text{Vol}(\bigcup_{R \in F_i} R)$ (i.e., volume of the union) makes optimization difficult. Therefore, for this step, $\hat{Q}(B_i)$ is defined as $\sum_{R \in F_i} \text{Vol}(R)$ (i.e., the sum of volumes) and $\sum_{B_i \in \mathcal{B}^{\text{Leaves}}} \hat{Q}(B_i)$ is minimized, instead. This objective function is more tractable, and the optimal solution under $\hat{Q}(B_i)$ approximates the optimal solution under $Q(B_i)$ within a factor of α_i. This objective function also discourages choosing overlapping rectangles for filters. In other words, we minimize

$$\sum_{B_i \in \mathcal{B}^{\text{Leaves}}, R_k \in \mathcal{R}} \text{Vol}(R_k) y_{ik},$$

subject to the following constraints:

(C1) [Filter complexity] Each broker B_i is assigned a filter consisting of at most α_i rectangles:

$$\sum_{R_k \in \mathcal{R}} y_{ik} \leq \alpha_i \quad \forall B_i \in \mathcal{B}^{\text{Leaves}}.$$

(C2) [Assignment and latency] Each subscriber is assigned to at least one broker meeting the latency constraint:

$$\sum_{B_i \in \mathcal{B}_j} x_{ij} \geq 1 \quad \forall S_j \in \mathcal{S}_a.$$

(C3) [Load balance] The load balance factor is at most $\overline{\beta}$:

$$\sum_{S_j \in \mathcal{S}_b} x_{ij} \leq \overline{\beta} \kappa_i |\mathcal{S}_b| \quad \forall B_i \in \mathcal{B}^{\text{Leaves}}.$$

(C4) [Nesting] A subscriber can only be assigned to a broker whose filter contains it:

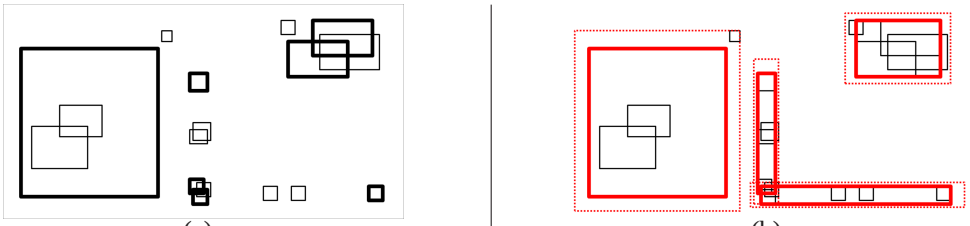
$$\sum_{R_k \in \mathcal{R}_j} y_{ik} \geq x_{ij} \quad \forall S_j \in \mathcal{S}_a, \forall B_i \in \mathcal{B}_j.$$

By relaxing the values of Boolean variables to be real numbers (i.e., $x_{ij}, y_{ik} \in [0, 1]$), the above mixed integer program can be reduced to an LP. Using an LP algorithm, the optimal fractional solution is computed, and then randomized rounding [114] is applied to construct a solution to the filter-assignment problem. Specifically, for each y_{ik}, suppose $\hat{y}_{ik}$ is its value in the optional fractional solution. y_{ik} is set to 1 with probability $1 - (1 - \hat{y}_{ik})^{\ln |\mathcal{S}_a|}$, or 0 otherwise. The resulting filter assignment is $\Phi = \{\varphi_1, \ldots, \varphi_n\}$, where $\varphi_i = \{R_k \mid y_{ik} = 1\}$.

Before returning Φ as a preliminary filter assignment, LPRelax further verifies whether Φ *covers* $\mathcal{S}_a$. More precisely, a subscriber S_j is said to be *covered* by a filter assignment Φ if there exists a broker B_i with assigned filter φ_i such that S_j's interest s_j is contained in one of the rectangles of φ_i, and the assignment of S_j to B_i satisfies the latency constraint for S_j. A set of subscribers is *covered* by a filter assignment if every subscriber in the set is covered. If it happens that Φ does not cover $\mathcal{S}_a$, randomized rounding is simply performed again for the y_{ik}'s to generate a new Φ. Each round of randomized rounding produces a Φ covering $\mathcal{S}_a$ with probability at least $\exp(-1)$ (see Theorem 27 presented later).

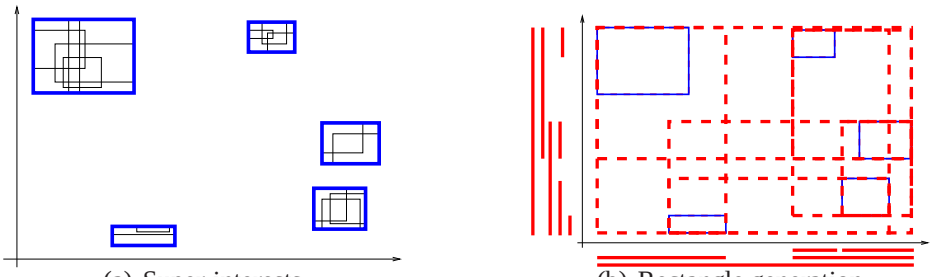
Remark Because of rounding, φ_i may have more than α_i rectangles; this violation is fine for now—recall from the beginning of Section 6.4 that the goal of this first step is not the *final* filter assignment, but a good, preliminary assignment for the remaining steps; Section 6.4.3 will fix such violations.

Note that randomized rounding could also be applied to x_{ij}'s and obtain a subscriber assignment for $\mathcal{S}_a$, but the resulting assignment may violate constraints due to rounding, and it is not the goal of this step of the algorithm.



(a) (b)

FIGURE 6.4: (a) Coreset members are drawn with thick outlines; (b) filters covering the coreset (thick rectangles) are ε-expanded (dotted lines) to cover all subscribers.



(a) Super-interests (b) Rectangle generation

FIGURE 6.5: Illustration of candidate filter generation.

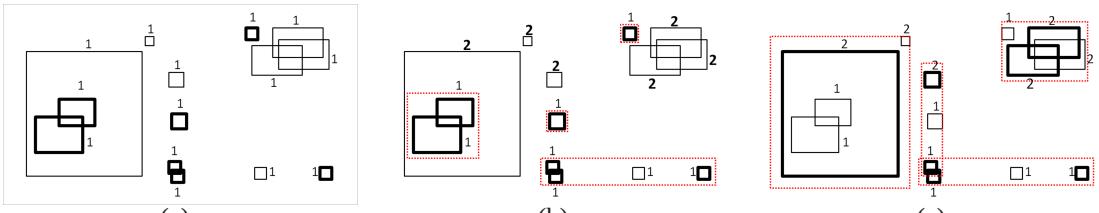
6.4.1.2 *Subscriber Sampling*

If all subscribers are inputted as $\mathcal{S}_a$ and $\mathcal{S}_b$ to LPRelax, the size of LP in Section 6.4.1.1 will be too large even for a moderate number of subscribers. Therefore, this section presents a method to reduce the number of subscribers to input to LPRelax. This method combines two ideas:

- *Coreset*: For a wide range of geometric optimization problems, there exists a small subset (*coreset*) of the input objects such that the solution for this subset is a good approximation of the solution for the entire input [6]. This chapter shows that for filter assignment, a small coreset of $\mathcal{S}$ exists and can be computed quickly.

- *Iterative reweighted sampling*: This idea has been previously used for problems such as linear programming [47], set cover [30], and computing coresets [10]. This chapter applies it to coreset computation for filter assignment.

We begin with a few definitions. For a rectangle $R = \prod_{i=1}^{d}[l_i, h_i]$, the *$\varepsilon$-expansion* of R, denoted by $(1+\varepsilon)R$, is $\prod_{i=1}^{d}[l_i - \varepsilon(h_i - l_i)/2, h_i + \varepsilon(h_i - l_i)/2]$. Similarly, the *$\varepsilon$-expansion* of



(a) (b) (c)

FIGURE 6.6: Three steps of iterative reweighted sampling: The weight of each subscriber is initially one (see (a)). Choose a subset $\mathcal{S}_a$ (thick rectangles); find a filter assignment Φ of $\mathcal{S}_a$; If the expansion of Φ (dotted rectangles) covers $\mathcal{S}$ (see case (c)), return Φ. Otherwise, double the weight of all $S \in \mathcal{S}$ not covered by the expansion of Φ (see case (b)).

a filter $\varphi = \{R_1, \ldots, R_\alpha\}$ is $(1+\varepsilon)\varphi = \{(1+\varepsilon)R_1, \ldots, (1+\varepsilon)R_\alpha\}$. Let $\Phi = \{\varphi_1, \cdots, \varphi_n\}$ be a filter assignment to $\mathcal{B}^{\text{Leaves}}$, with φ_i being the filter associated with B_i, and let $(1+\varepsilon)\Phi = \{(1+\varepsilon)\varphi_1, \ldots, (1+\varepsilon)\varphi_n\}$. A coreset $\mathcal{Q} \subseteq \mathcal{S}$ is called an *ε-certificate* if, for any filter assignment Φ that covers $\mathcal{Q}$, $(1+\varepsilon)\Phi$ covers $\mathcal{S}$ (recall the definition of "cover" from Section 6.4.1.1). The notion of coreset is illustrated in Figure 6.4. Lemma 28 in Section 6.4.4 shows that there is always an ε-certificate whose size is independent of $|\mathcal{S}|$ (although the worst case bound is exponential in $|\mathcal{B}^{\text{Leaves}}|$). The size of an ε-certificate is likely to be much smaller in practice—as evident from the empirical results.

The remainder of this section is devoted to describe FilterAssign$(\mathcal{B}^{\text{Leaves}}, \mathcal{S})$ (Algorithm 9), for computing a preliminary filter assignment using these ideas. If there exists an ε-certificate of size g, an iterative reweighted sampling scheme can compute an ε-certificate of size $O(g \ln g)$ in $O(g \ln |\mathcal{S}|)$ iterations (Lemma 30 at the end of the chapter). Without knowing g in advance, FilterAssign performs an exponential search on g, running $O(g \ln |\mathcal{S}|)$ iterations for a fixed value of g and then doubling it.

Each stage of the search targets a specific g and consists of multiple *valid* iterations.[2] FilterAssign maintains a weight for each subscriber in $\mathcal{S}$, initialized to 1 at the beginning of the stage. Each iteration chooses a random subset $\mathcal{Q} \subseteq \mathcal{S}$ of size $O(g \ln g)$, where each subscriber is chosen with

[2] This validity condition is needed to establish the termination condition of an iteration (Line 14 of Algorithm 9). A *valid* iteration is one where the ratio of the total weight of uncovered subscribers to that of all subscribers is no more than $1/8$. By random sampling theory (Lemma 31 at the end of the chapter), an iteration is valid with probability at least $1/2$, so an iteration can simply be re-done until it is valid.

probability proportional to its weight. A filter assignment for $\mathcal{Q}$ is computed using a helper procedure FilterAssignHelper described below. If the procedure finds an assignment Φ (by calling LPRelax), FilterAssign checks whether $(1+\varepsilon)\Phi$ covers the entire $\mathcal{S}$. If yes, FilterAssign stops and returns $(1+\varepsilon)\Phi$. Otherwise, FilterAssign doubles the weight of each subscriber not covered by Φ, and begin a new iteration. An example is shown in Figure 6.6. If the number of valid iterations for the stage exceeds $4g\ln(|\mathcal{S}|/g)$, FilterAssign concludes that the ε-certificate has size larger than g (by Lemma 30), and FilterAssign moves on to the next stage.

FilterAssignHelper, invoked by FilterAssign's inner loop, further prepares the input for LPRelax and calls it. The ε-certificate $\mathcal{Q}$ that we look for in FilterAssign is intended for the problem of covering $\mathcal{S}$, but since LPRelax considers coverage and load balance jointly, FilterAssignHelper must ensure that the input to LPRelax properly reflects the properties of $\mathcal{S}$ relevant to load balancing. To this end, FilterAssignHelper chooses a random subset $\mathcal{S}_b \subseteq \mathcal{S}$ of size proportional to $|\mathcal{B}^{\text{Leaves}}|$ (in the experiments, $10|\mathcal{B}^{\text{Leaves}}|$ is used for the practical sizes of $\mathcal{B}^{\text{Leaves}}$). FilterAssignHelper calls LPRelax with $\mathcal{S}_a = \mathcal{Q} \cup \mathcal{S}_b$, and $\mathcal{R} =$ FilterGen$(\mathcal{S}_a)$, where FilterGen is the candidate filter generation procedure to be described in Section 6.4.1.3. To guard against the small possibility that a random choice of $\mathcal{S}_b$ makes the otherwise feasible optimization problem infeasible, FilterAssignHelper repeats with a new choice of $\mathcal{S}_b$ (up to a few times) if LPRelax fails to find a feasible solution.

6.4.1.3 Candidate Filter Generation.

This section describes the procedure FilterGen for constructing the set $\mathcal{R}$ of rectangles to be used by LPRelax to form filters. Without loss of generality, let $\mathcal{S} = \{S_1, \cdots, S_m\}$ denote the set of subscribers given as input to FilterGen (in reality, a subset may be given instead), and let s_i denote S_i's interest (a rectangle in $\mathbb{R}^d$). Each rectangle in $\mathcal{R}$ is intended to contain a subset of $\mathcal{S}$. There are $\Omega(m^{2d})$ rectangles, each of which contains a distinct subset.[3] However, this many rectangles make

[3] This lower bound is tight. In the case of $d = 1$, each interest is an interval. Any interval I containing a subset of the m intervals can be shrunk so that the endpoints of I coincide with the endpoints of some of the m intervals. Hence, there are $O(m^2)$ candidate intervals. Generalizing this argument to higher dimensions, $O(m^{2d})$ candidate rectangles can be generated in $\mathcal{R}$.

LPRelax impractical.

FilterGen takes two steps (see Figure 6.5) to ensure that $\mathcal{R}$ is small yet provides good coverage. The first step replaces the input subscriber interests with a set $\Xi = \{\xi_1, \cdots, \xi_k\}$ of k *super-interests*, where k is proportional to the number of brokers (In the experiments, k is set to $5|\mathcal{B}^{\text{Leaves}}|$). These super-interests are obtained by partitioning $\mathcal{S}$ into k

clusters and choosing the minimum enclosing box (MEB) of the subscriber interests in each cluster. This clustering is done in a joint network-event space for two reasons: 1) It captures the correlation between geographical and topical concentration of interests; 2) compared to a two-stage clustering (clustering first in one space and then consider another space), it is easier to control the size of $\mathcal{R}$ under a single-stage clustering. In the second step, instead of generating $O(k^{2d})$ rectangles, a hierarchical procedure is used to generate fewer rectangles. The intuition is that if latency and load balancing constraints are not too tight, there is flexibility in assigning subscribers to brokers and each broker would handle subscribers with similar interests. The hierarchical procedure aims at generating filters for the clusters of interests on various levels of granularity. Now the two steps are described in more detail.

For clustering, a subscriber S with coordinate $(x_1, \ldots, x_t)$ in the network space $\mathbb{N} = \mathbb{R}^t$ and interest $\prod_{i=1}^{d}[l_i, h_i]$ in the event space $\mathbb{E} = \mathbb{R}^d$ can be mapped to a point

$$(x_1, \ldots, x_t, l_1, \ldots, l_d, h_1, \ldots, h_d)$$

in $\mathbb{R}^{t+2d}$. Let $\mathcal{P} = \{s_j^* \mid j \in [1, m]\}$ be the resulting set of m points in $\mathbb{R}^{t+2d}$. $\mathcal{P}$ is partitioned into k clusters using the k-means algorithm. Let $\mathcal{P}_1, \ldots, \mathcal{P}_k$ be the clusters returned by the algorithm. For each $\mathcal{P}_j$, let $\xi_j \subseteq \mathbb{E}$ be the MEB of subscriber interests corresponding to the points in $\mathcal{P}_j$. The desired set of super-interests is $\Xi = \{\xi_1, \ldots, \xi_k\}$.

In the second step, for each dimension $i \in [1, d]$, FilterGen constructs a set $\mathcal{J}_i$ of intervals lying on the x_i-axis. $\mathcal{R}$ is set to be the Cartesian product of these sets, i.e., $\mathcal{R} = \{J_1 \times \cdots \times J_d \mid \forall i \in [1, d] : J_i \in \mathcal{J}_i\}$. It thus remains to describe the construction of $\mathcal{J}_i$. Let $\mathcal{I}_i$ be the set of k intervals that are the projection of Ξ onto the x_i-axis. Let Δ be the length of the smallest interval containing $\mathcal{I}_i$, and let δ be the length of the smallest interval in $\mathcal{I}_i$. For $1 \leq j \leq \lceil \log_2(\Delta/\delta) \rceil$, let $\ell_j = 2^j\delta$. (If Δ/δ is large, FilterGen chooses ℓ_j's more carefully.) For each j, let $\mathcal{I}_{ij} \subseteq \mathcal{I}_i$ be the

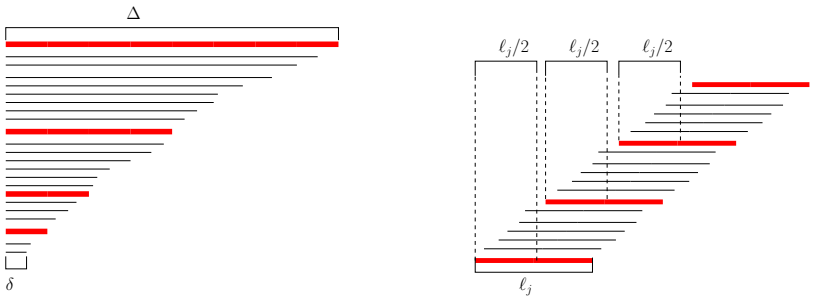


FIGURE 6.7: Two main ideas for the rectangle generation step: (a) Consider only $\log_2(\Delta/\delta)$ different lengths, (b) No two intervals of length ℓ_j overlap by more than $\ell_j/2$.

set of intervals of length at most $\ell_j/2$. FilterGen generates a set of intervals $\mathcal{J}_{ij}$ of length at most ℓ_j such that every interval of $\mathcal{I}_{ij}$ is contained by some interval in $\mathcal{J}_{ij}$, and no two intervals in $\mathcal{J}_{ij}$ overlap by more than $\eta\ell_j$ (In the experiments, η is set to $1/2$). Figure 6.7 illustrates the idea.

To avoid two intervals in $\mathcal{J}_{ij}$ overlapping by more than $\eta\ell_j$, let $\mathcal{L}$ be the set of left endpoints of intervals in $\mathcal{I}_{ij}$, sorted in increasing order. $\mathcal{L}$ is scanned from left to right and do the following. The first point, say p, of $\mathcal{L}$ is taken, and all the points from $\mathcal{L}$ that are within distance $(1-\eta)\ell_j$ from p are removed. Let J be the interval of length ℓ_j with p as its left endpoint. J is shrunk to the smallest possible interval such that it still contains the same subset of intervals in $\mathcal{I}_{ij}$. Then J is added to $\mathcal{J}_{ij}$ and the above step is repeated, until $\mathcal{L}$ becomes empty, at which point $\mathcal{J}_{ij}$ is added to $\mathcal{J}_i$ and move on to the next j. In the worst case, $|\mathcal{J}_i| = O(k \log_2 \Delta/\delta)$, but in practice it is expected to be closer to $O(k)$ or even smaller. Hence, the size of the filter candidate set is $O(k^d)$, but it can be further reduced by working in high dimension directly if the dimensionality of $\mathbb{E}$ is large. FilterGen shrinks each rectangle $R \in \mathcal{R}$ to the MEB of subscriber interests contained by R and returns $\mathcal{R}$ to FilterAssignHelper.

6.4.2 Subscription Assignment

The second step of SLP_1 takes as input the preliminary filter assignment Φ produced by FilterAssign in Section 6.4.1, and computes the subscriber assignment $\Sigma : \mathcal{S} \rightarrow \mathcal{B}^{\text{Leaves}}$, for the entire set of subscribers. Since the filters are already given, minimizing bandwidth is not a concern here; instead, the focus is concentrated on load balance while ensuring that subscribers are only assigned to

brokers that *cover* them (recall the definition of "cover" from Section 6.4.1.1, which considers both nesting and latency constraints). Also, recall from Section 6.2 that $\overline{\beta}$ and $\beta_{\max}$ are user-defined desired and maximum load balance factors (lbfs), resp.; the goal is to find a Σ whose lbf is no more than $\overline{\beta}$, or else, close to $\overline{\beta}$ and no more than $\beta_{\max}$.

The computation of Σ is formulated as a max-flow problem. A bipartite graph $G = (V, E)$ is constructed, where $V = \mathcal{S} \cup \mathcal{B}^{\text{Leaves}} \cup \{s, t\}$, $E = E_1 \cup E_2 \cup E_3$, $E_1 = \{(s, B) \mid B \in \mathcal{B}^{\text{Leaves}}\}$, $E_2 = \{(S, t) \mid S \in \mathcal{S}\}$, and $E_3 = \{(B_i, S_j) \mid B_i \text{ covers } S_j\}$. The capacity of every edge in $E_2 \cup E_3$ is set to 1, and the capacity of an edge (s, B_i) in E_1 to $\lfloor \beta \kappa_i |\mathcal{S}| \rfloor$. Initially, $\beta = \overline{\beta}$, but it may increase over time to $\beta_{\max}$.

The maximum flow is computed from s to t. Let f be the value of the maximum flow. If $\mathsf{f} = |\mathcal{S}|$, then every subscriber in $\mathcal{S}$ is assigned to a broker, which can be identified by the edge into the subscriber with flow of 1. The resulting subscriber assignment, which by construction has a lbf of no more than β, is returned. If $\mathsf{f} < |\mathcal{S}|$ and $\beta = \beta_{\max}$, a conclusion is drawn that the load balance constraint is too tight, and SLP_1 stops. If $\mathsf{f} < |\mathcal{S}|$ and $\beta < \beta_{\max}$, the value of β is increased by a small factor, update the capacity of the edges in E_1, and recompute the maximum flow from s to t. Depending on the maximum flow algorithm employed, as an optimization, the current flow can be reused as the starting flow for the increased value of β [73].

6.4.3 Filter Adjustment

The third and last step of SLP_1 further adjusts the preliminary filter assignment $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ made by FilterAssign. Based on the subscriber assignment $\Sigma : \mathcal{S} \to \mathcal{B}^{\text{Leaves}}$ made by the second step, this step opportunistically tightens the filters, and enforces the filter complexity constraint (that each φ_i consists of no more than α_i rectangles). Consider each broker B_i with preliminary filter φ_i. Let $\mathcal{S}_i \subseteq \mathcal{S}$ be the set of subscribers assigned to B_i. We want to replace φ_i by F_i, a set of no more than α_i rectangles, such that $\bigcup_{S_j \in \mathcal{S}_i} s_j \subseteq \bigcup_{R \in F_i} R$ and $Q(B_i) = \text{Vol}(\bigcup_{R \in F_i} R)$ is minimized. The problem is NP-hard [27] in general, so the following simple heuristic is used.

The subscriber interests associated with $\mathcal{S}_i$ are partitioned into α_i groups, using the same clustering technique as super-interest generation in Section 6.4.1.3 but ignoring the network space.

This partitioning gives us a filter with α_i rectangles, each of which is the MEB of the interests in a group.

If φ_i has no more than α_i rectangles, φ_i is also adjusted as follows. Each subscriber in $\mathcal{S}_i$ is assigned to a rectangle in φ_i containing its interest (if there are multiple rectangles, one is chosen arbitrarily). Then, each rectangle in φ_i is replaced by the MEB of the interests of the subscribers assigned to it. The resulting volume of φ_i often decreases. Between φ_i and the filter generated by the clustering technique above, the one with the smaller volume is chosen to be F_i.

After processing all filters, Σ and $\mathcal{F} = \{\bigcup_{R \in F_1} R, \ldots, \bigcup_{R \in F_n} R\}$ are returned as the final result. This completes the description of SLP_1.

6.4.4 Solution Quality

We begin with a discussion of the solution quality of FilterAssign, the first step of SLP_1. Recall the mixed integer program described in Section 6.4.1.1. Let $\mathrm{OPT}_{\mathrm{LP}}(\mathcal{B}^{\mathrm{Leaves}}, \mathcal{R}, \mathcal{S}_a, \mathcal{S}_b)$ denote the value of the objective function ($\sum_{B_i \in \mathcal{B}^{\mathrm{Leaves}}} \hat{Q}(B_i)$) for the optimal LP factional solution to this program (by allowing the values of Boolean variables to be real). The following theorem bounds the quality of the solution produced by LPRelax (Section 6.4.1.1) in terms of $\mathrm{OPT}_{\mathrm{LP}}$.

Theorem 27 (Solution quality of LPRelax)**.** *LPRelax*($\mathcal{B}^{\mathrm{Leaves}}$, $\mathcal{R}, \mathcal{S}_a, \mathcal{S}_b$) *returns a filter assignment with the following properties. i) The expected value of the objective function is at most* $\ln|\mathcal{S}_a|\mathrm{OPT}_{\mathrm{LP}}(\mathcal{B}^{\mathrm{Leaves}}, \mathcal{R}, \mathcal{S}_a, \mathcal{S}_b)$*. ii) The expected filter complexity of* B_i *is no more than* $\ln|\mathcal{S}_a|\alpha_i$*. Furthermore, with probablity at least* $1/e$*, a subscriber assignment can be found such that: iii) it satisfies the nesting constraint with respect to the returned filter assignment; iv) it satisfies the latency constraint; and v) its expected lbf, with respect to the subscribers in* $\mathcal{S}_b$*, is at most* $\ln|\mathcal{S}_a|\overline{\beta}$.

From Theorem 27 above, we see that for LPRelax's solution, its expected quality can be a factor of $\ln|\mathcal{S}_a|$ worse than $\mathrm{OPT}_{\mathrm{LP}}$, and its expected filter complexity can exceed the maximum allowed by a factor of $\ln|\mathcal{S}_a|$ as well. Fortunately, as the following lemma shows, the size of an ε-certificate is independent of $|\mathcal{S}|$; therefore, $|\mathcal{S}_a|$ is likely much smaller than $|\mathcal{S}|$, so the blow-up factor is closer to a small constant—as evident from the empirical results.

Lemma 28 (Size of coreset for filter assignment)**.** *There exists an ϵ-certificate $\mathcal{Q} \subseteq \mathcal{S}$ of size $O((n \ln(\Delta/\epsilon))^{2dn \max(\boldsymbol{\alpha})})$, where Δ is proportional to the ratio of the volume of $\mathrm{MEB}(\mathcal{S})$ to the volume of the smallest subscriber interest.*

$\mathrm{OPT}_{\mathrm{LP}}(\mathcal{B}^{\mathrm{Leaves}}, \mathcal{R}, \mathcal{S}_a, \mathcal{S}_b)$ provides a lower bound for the value of the objective function for the optimal solution to the mixed integer problem with the same inputs. Furthermore, since the optimal filter assignment for $\mathcal{S}$ is also a filter assignment for $\mathcal{S}_a \subseteq \mathcal{S}$, $\mathrm{OPT}_{\mathrm{LP}}$, optimal with respect to $\mathcal{S}_a$, must be a lower bound for the optimal solution with respect to $\mathcal{S}$. However, restricting the set of rectangles $\mathcal{R}$ to be considered for filters, as done in Section 6.4.1.3, can increase $\mathrm{OPT}_{\mathrm{LP}}$ and make it no longer a lower bound. Note that the two steps in candidate filter generation are orthogonal. Given the set of super-interests provided by the first step, the pruning of filters in the second step only degrades $\mathrm{OPT}_{\mathrm{LP}}$ by a constant factor because, as the following lemma shows, for any rectangle R excluded from the candidate set $\mathcal{R}$, there exists $R' \in \mathcal{R}$ such that $R \subseteq R'$ and $\mathrm{Vol}(R) \approx \mathrm{Vol}(R')$.

Lemma 29 (Goodness of candidate filters)**.** *Let $\mathcal{R}^*$ be the set of $O(k^{2d})$ rectangles, where each rectangle is the minimum enclosing box of a subset of the k subscriber interests. Let $\mathcal{R}$ be the set of candidate rectangles returned by FilterGen. For each rectangle $R \in \mathcal{R}^* \setminus \mathcal{R}$, there exists a rectangle $R' \in \mathcal{R}$, such that $R \subset R'$ and $\mathrm{Vol}(R') \leq 4^d \mathrm{Vol}(R)$.*

However, the blow-up cannot be bounded due to the super-interest clustering step; it is a necessary trade-off between complexity of the algorithm and optimality of its solution. Nonetheless, if this first step of candidate filter generation is skipped (i.e. every subscriber interest is a super-interest) and only the second is applied, then $\mathrm{OPT}_{\mathrm{LP}}$ obtained with the resulting $\mathcal{R}$ still matches the lower bound for the optimal solution up to a small constant factor.

In sum, FilterAssign produces a preliminary filter assignment that has provably good bandwidth and bounded filter complexity (by (i) and (ii) in Theorem 27 and discussion above) and can lead to a good subscriber assignment (by (iii), (iv) and (v) in Theorem 27).

Given this preliminary filter assignment, the subscriber assignment step in Section 6.4.2 further optimizes load balancing. The entire $\mathcal{S}$ is considered for load balancing by this step (as opposed

to only $\mathcal{S}_b$ by FilterAssign). The max-flow algorithm is guaranteed to find the most load-balanced subscriber assignment possible.

Finally, the filter adjustment step in Section 6.4.3 enforces the filter complexity of each broker, now using $Q(B_i)$ (volume of union, as introduced in Section 6.2) instead of $\hat{Q}(B_i)$ (sum of volumes, as introduced in Section 6.4.1.1) in the objective function. When divided by the maximum filter complexity, a lower bound for the optimal solution under $\hat{Q}(B_i)$ serves as a lower bound for the optimal solution under $Q(B_i)$.

6.5 Multi-Level SA

This section describes an algorithm for SA called SLP when the broker tree $\mathcal{T}$ has multiple levels of brokers. One possible approach is to first run the one-level algorithm SLP_1 (Section 6.4) over all leaf brokers, and then compute the filters at the interior nodes of $\mathcal{T}$ in a bottom-up manner. This approach has two drawbacks. First, sibling brokers in $\mathcal{T}$ may be assigned subscribers with very different interests, forcing a large filter at their parent which consumes a lot of bandwidth. Second, solving SLP_1 on a large set of brokers is computationally expensive. In practice, broker trees often follow the topology of the underlying network, so a top-down hierarchical approach will be effective.

The multi-level algorithm SLP works by recursively applying the one-level algorithm SLP_1 to subtrees in $\mathcal{T}$ in a top-down manner. At each non-leaf broker B of $\mathcal{T}$, SLP_1 is invoked to distribute the subscribers among B's children, deciding in which subtree of B each subscriber will be assigned. SLP then recursively processes each child with the set of subscribers assigned to the corresponding subtree.

To invoke SLP_1 over a set of non-leaf sibling brokers, still need to address the issues of determining appropriate latency and load balance constraints for assigning a subscriber to these brokers—recall from Section 6.2 that the actual latency to a subscriber depends on its leaf broker assignment, which has not been made yet because of top-down processing; the load balance constraints have only been defined for leaf brokers. These two issues are addressed below.

Determining Latency Constraints. Suppose the multi-level algorithm SLP has passed a subscriber S_j to the subtree rooted at a non-leaf broker B. For the purpose of running SLP_1 over B's children, SLP needs to determine, for each child broker B' of B, whether assigning S_j to B' satisfies the latency constraint. Consider $\mathsf{Leaves}(B')$, the set of leaf brokers in the subtree rooted at B'. Let $\gamma_j(B') \in [0,1]$ denote the fraction of leaf brokers in $\mathsf{Leaves}(B')$ that would satisfy the latency constraint for S_j if S_j is eventually assigned to them. A threshold $\overline{\gamma}$ is set such that $\gamma_j(B') \geq \overline{\gamma}$ if and only if assigning S_j to B' satisfies the latency constraint when running SLP_1 over B's children. The choice of the threshold reflects a trade-off: A high $\overline{\gamma}$ could severely limit the choices of subtrees to which S_j can be assigned, making it difficult to distribute subscribers evenly among the subtrees. A low $\overline{\gamma}$, on the other hand, means that S_j could be assigned to a subtree with few leaf brokers satisfying the latency constraint for S_j, making it difficult to distribute subscribers evenly within the subtree. $\overline{\gamma}$ is set to $1/2$ to balance these two concerns.

In the event that $\gamma_j(B') < \overline{\gamma}$ for every child B' of B, $\overline{\gamma}$ is lowered by a factor of two and try again, until $\gamma_j(B') \geq \overline{\gamma}$ for at least one B'. This procedure ensures that S_j can be assigned to a subtree even under stringent latency constraints.

Determining Load Balance Constraints. First, for each child broker B' of broker B, $\kappa(B')$, the capacity fraction of B', is set to be $K(B')/K(B)$, where $K(B) = \sum_{B_i \in \mathsf{Leaves}(B)} \kappa_i$ is the sum of capacity fractions of leaf brokers in the subtree rooted at B. It is easy to see that the capacity fractions of B's children sum up to exactly 1. If B is passed $m(B)$ subscribers to handle, the *locally perfectly balanced load* for child B' would be $\kappa(B') \cdot m(B)$.

Some care is required for determining $\overline{\beta}(B)$ and $\beta_{\max}(B)$, the desired and maximum lbfs (resp.) for running SLP_1 over B's children. Setting these lbfs to their user-specified global counterparts, i.e., $\overline{\beta}(B) = \overline{\beta}$ and $\beta_{\max}(B) = \beta_{\max}$, does not work. The reason is that, for a path of length ℓ to a leaf broker B_i, if the multi-level algorithm SLP allows the number of subscribers passed to every broker to exceed its locally perfectly balanced load by a factor of β, then the total excess along the path would accumulate to a factor of β^ℓ over $\kappa_i|\mathcal{S}|$. Therefore, the following method is used instead to assign $\overline{\beta}(B)$ and $\beta_{\max}(B)$. Note that if the load is perfectly balanced

globally, B should have been passed $K(B) \cdot |\mathcal{S}|$ subscribers. Suppose $m(B)$ is the actual number of subscribers given to B by SLP. $\overline{\beta}(B)$ is set to $(\overline{\beta} / \frac{m(B)}{K(B) \cdot |\mathcal{S}|})^{1/\ell}$ and $\beta_{\max}(B) = (\beta_{\max} / \frac{m(B)}{K(B) \cdot |\mathcal{S}|})^{1/\ell}$, where ℓ is the path length from B to leaf brokers.[4] Effectively, this method adjusts the lbfs dynamically as SLP recurses down $\mathcal{T}$, accounting for the variable amount of excess load generated by each step.

Remark SLP targets dissemination trees with large fan-out values but few number of levels. If the height of a dissemination tree is large, solving subscriber assignment level-by-level is not a right approach.

6.6 Evaluation

Other Algorithms Tested. Other algorithms are also considered for comparison with Gr, Gr*, SLP$_1$, and SLP. The first one is a variant of Gr that ignores latency. (Note that it is less sensible to ignore load balance, because there would be a strong incentive to assign every subscriber to the same broker.)

- ***Online Greedy without Latency Consideration*** *(Gr$_{\neg l}$)*. This algorithm works exactly like Gr, except that it drops the latency constraint in defining candidate broker sets. The answer produced by Gr$_{\neg l}$ is useful in understanding how latency constraints affect attainable bandwidth.

Other algorithms that ignore bandwidth and instead focus on some other performance metrics, are considered additionally. As we will see, like Gr$_{\neg l}$, these algorithms do poorly on the metrics they ignore, but they help illustrate the importance of considering multiple metrics jointly in optimization.

- ***Closest Broker without Load Balance*** *(Closest$_{\neg b}$)*. This algorithm resembles the one in [13]. It assigns each subscriber to its closest leaf broker in the network space (hence minimizing last-hop latency). Ties are broken arbitrarily.

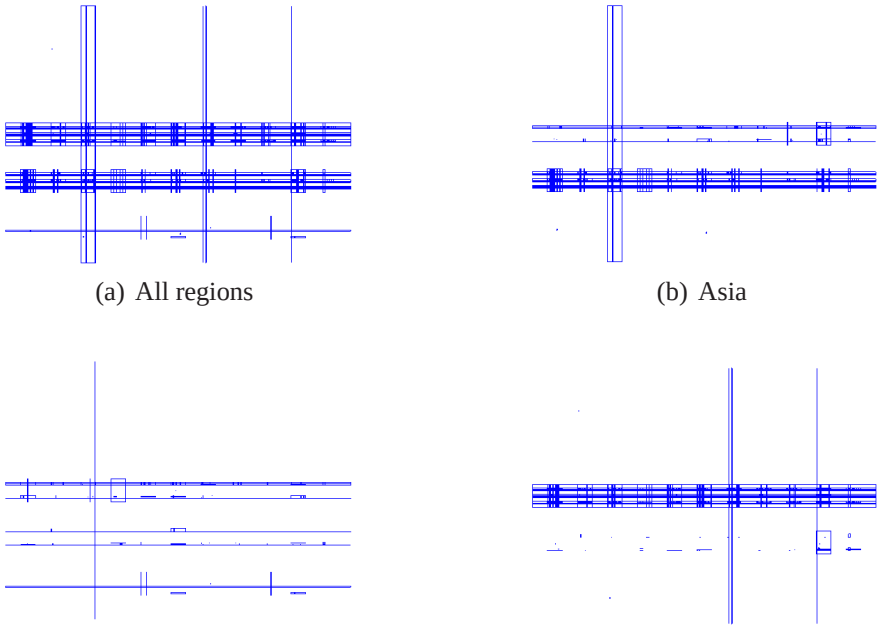
[4] For simplicity of presentation, this setting assumes that $\mathcal{T}$ is height-balanced; i.e., all leaves are an equal number of hops away from the root. Generalization to the unbalanced case is straightforward.

- ***Closest Broker*** *(Closest).* Like Closest$_{\neg b}$, this algorithm assigns each subscriber to its closest leaf broker. However, once a broker has already been assigned the maximum number of subscribers allowed by the user-specified maximum lbf $\beta_{\max}$, Closest drops it from further consideration.

- ***Best Load-Balanced Assignment*** *(Balance).* This algorithm finds the assignment with the best possible lbf (possibly less than the user-specified desired lbf $\overline{\beta}$) by solving a max-flow problem. The graph construction is a variant of the one in Section 6.4.2.

Workloads. The above algorithms are evaluated using three sets of workloads. As discussed in Section 6.1, it is important to base evaluation on realistic workloads, but they have been difficult to find. This issue is addressed in [121] by developing a workload generator based on publicly available statistics on Google Groups. Extrapolating from these statistics, the generator produces a baseline workload consistent with them, and can generate additional workloads that deviate in meaningful ways from the baseline. Multiple workloads produced by this generator (collectively referred to as ***workload set #1***) are used for evaluation. The network locations are mapped to points in $\mathbb{N} = \mathbb{R}^5$, and the subscriber interests are rectangles in $\mathbb{E} = \mathbb{R}^2$. Two workload factors—*IS*, interest skewness in terms of popularity, and *BI*, number of broad interests (i.e., large rectangles)—vary between the settings of L(ow) and H(igh). The baseline workload from Google Groups resembles (IS:H, BI:L). The distribution of subscribers across Asia, North America, and Europe is $4:1:4$. The distribution of brokers across the network space is set to be roughly the same as that of the subscribers.

The workload generator [121] uses data extracted from PlanetLab, which consists of 1019 nodes and 484 sites. The inter-node latency relationship is embed in a low-dimensional Euclidean space using [76]. The generator assumes that interest topics form a partially ordered set (poset). First, the 100 hottest topics are removed, because they correspond to extremely popular interests that are better handled by separate dissemination mechanisms such as broadcast. IS (interest skewness) is changed by *interest diffusion* [121], which adjusts the popularities of topics in the poset in a top-down fashion by balancing the popularities across subtopics to reduce their variance by a



(a) All regions (b) Asia

(c) North America (d) Europe

FIGURE 6.8: Interest distributions in $\mathbb{E}$ for (IS:H, BI:H).

user-specified factor. IS:L uses a factor of 55% while IS:H makes no adjustment. BI (broad interest) is adjusted by *interest generalization* [121], which increases the popularities of more general topics in the poset in a bottom-up fashion by propagating a fraction of the popularities of subtopics up. BI:L sets this fraction to 1% while BI:H sets it to 10%. For (IS:H, BI:H), Figure 6.8 illustrates the interest distributions in the event space by subscribers' geographic regions.

Workload set #2 is designed to reproduce those used for evaluation in [97, 92, 91], is based on observations of the RSS feed popularity. A total of 50 different interests are generated and their popularity follows a Zipf distribution with exponent 0.5. Each interest is mapped to a random unit square in $\mathbb{E}$. Given an interest, subscriber locations are drawn uniformly at random from 10 locations in $\mathbb{N}$. In this workload set, the subscriber interests are essentially topic-based, and no notion of "proximity" is captured in either the event space or the network space.

Workload set #3 is designed to mimic those used in ranked content-based publish/subscribe [79] and peer-to-peer overlay for content-based publish/subscribe [117, 26]. The event space is partitioned into 100 grid cells. The center of an interest is mapped to the center of one of the cells.

To create hot spots in $\mathbb{E}$, the cells are ranked in random order; the probability of picking a cell as an interest center follows a Zipf distribution with exponent 0.5. There is also a set of predefined interest widths. For each dimension, the width of an interest is chosen from this set according to a Zipf distribution with exponent 0.5. Each subscriber is randomly located at one of the network locations in $\mathbb{N}$; therefore, subscriber interests and locations are independent.

Problem Settings. Unless otherwise specified, the following settings are used for the SA problem, and the section shows how the parameters affect the results later. Filter complexity α is set to 3 for all brokers. Latency constraints are specified using a *maximum delay* of 0.3; the *delay* experienced by a subscriber S under a subscriber assignment Σ is defined to be $\delta/\Delta - 1$, where δ is the latency of the path in $\mathcal{T} \cup \Sigma$ from the publisher to S, and Δ is latency of the shortest path from the publisher to S through $\mathcal{T}$. For load balance constraints, all leaf brokers have equal capacity fractions. For workload set #1, the desired and maximum load balance factors, $\overline{\beta}$ and $\beta_{\max}$, are 1.5 and 1.8, respectively. For workload set #2, since the subscribers of an interest are restricted to a few network locations only, subscriber distribution is skewed in $\mathbb{N}$ due to interest skewness. Therefore, $\overline{\beta}$ and $\beta_{\max}$ are set to relatively relaxed values of 2.3 and 2.5, respectively. For workload set #3, since subscriber locations are completely random, $\overline{\beta}$ and $\beta_{\max}$ are tightened to 1.3 and 1.5, respectively.

The two greedy algorithms in Section 6.3 are compared with the algorithms described earlier in this section together with SLP_1 (for one-level broker networks) or SLP (for multi-level broker networks). The quality of a solution is measured in terms of total bandwidth, subscriber delays, and broker loads (i.e., number of subscribers assigned to each broker). For non-deterministic algorithms, the average (when applicable) of five runs is reported; deviation in results has been found to be insignificant.

6.6.1 Solution Quality for a One-Level Broker Network

In the following, there are $100{,}000$ subscribers to assign to 100 brokers attached directly to the publisher.

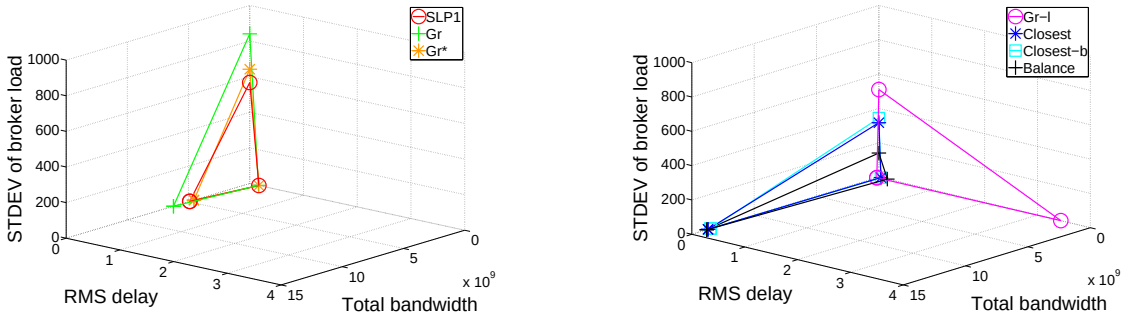


FIGURE 6.9: Overall comparison (one-level network, workload set #1).

Overall Comparison: Figures 6.9. To get a quick overview, the result quality of each algorithm on workload set #1 is plotted as a triangle whose vertices correspond to total bandwidth, root mean square (RMS) of delay across subscribers, and standard deviation (STDEV) of broker loads. The numbers reported are averaged over four workloads: (IS:L, BI:L), (IS:H, BI:L), (IS:L, BI:H), and (IS:H, BI:H).

The figure on the left shows that SLP_1 and $\mathsf{Gr}^\star$ do well in minimizing bandwidth while bounding delay and load balance. Gr is worse: not only it incurs higher bandwidth, but it also produces very unbalanced loads (while SLP_1 and $\mathsf{Gr}^\star$ stay right within the maximum lbf). In fact, for all four workloads, Gr fails to find a feasible solution that satisfies the load balance constraints; nonetheless, the best-effort solutions found by Gr are reported. Variants of Gr are also tried: whenever the greedy algorithm cannot assign a subscriber S_j (because all its candidate brokers are fully loaded), it randomly removes some subscribers from these brokers to make room for S_j, and either reassign the removed subscribers next, or append them to the list of subscribers to be processed later. These variants still failed to find feasible solutions, even when given longer time to run than SLP_1.

The figure on the right shows that algorithms that ignore one performance criterion or another do poorly. By failing to consider subscriber interests in the event space, $\mathsf{Closest}_{\neg\mathsf{b}}$, Closest, and Balance incur huge bandwidth. By ignoring latency constraints in the network space, $\mathsf{Gr}_{\neg\mathsf{l}}$ produces unacceptable delays. $\mathsf{Closest}_{\neg\mathsf{b}}$ has okay load balance in this case only because the broker and subscriber distributions are similar; in general $\mathsf{Closest}_{\neg\mathsf{b}}$'s load imbalance can be arbitrarily bad.

Similar results are observed for workload sets #2 and #3, as shown in Figures 6.10, and 6.11, respectively.

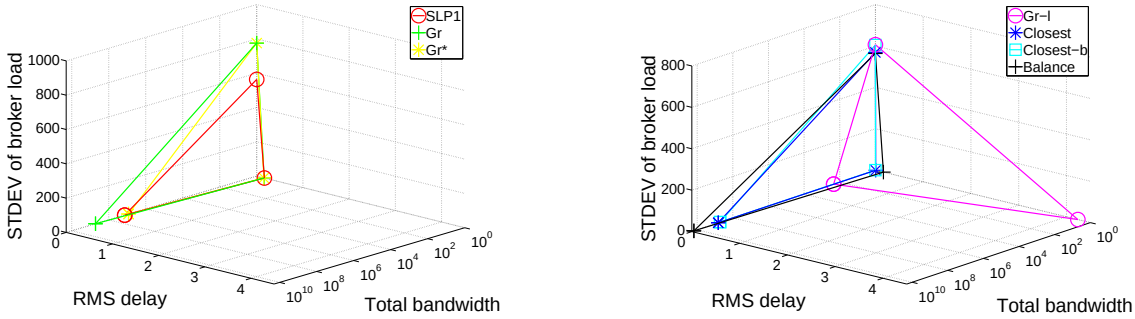


FIGURE 6.10: Overall comparison (one-level network, workload set #2).

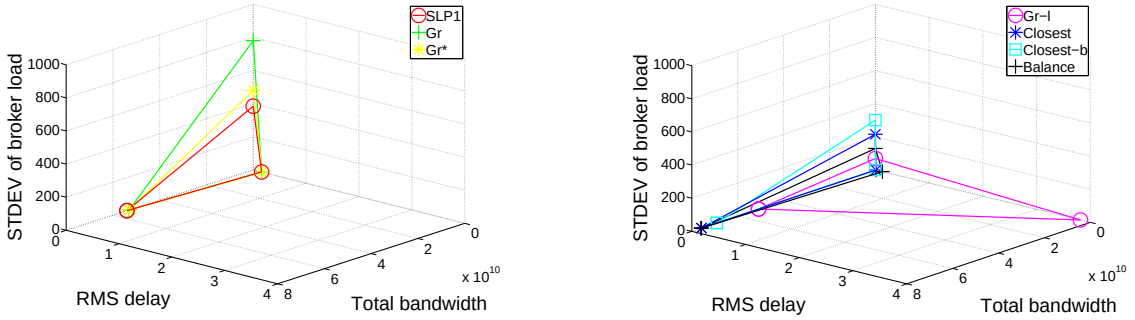


FIGURE 6.11: Overall comparison (one-level network, workload set #3).

One question that is set out to answer with these experiments is whether, in practice, the solution could be used to a more tractable optimization problem that ignores some constraints as a (lower-bound) yardstick for gauging the quality of the solution to the full optimization problem. Here it is clear that $\mathsf{Gr}_{\neg \mathsf{l}}$ is not a good yardstick—compared with the other algorithms, its bandwidth is just too low and too unrealistic to serve as a meaningful yardstick.

But then, how could a conclusion be obtained that a solution is "good enough" with respect to the optimal? The solution of SLP_1, though not guaranteed to be optimal, serves as a reasonable indicator because of SLP_1's theoretical properties. Next, we will see how a by-product of running SLP_1, namely the LP fractional solution (Section 6.4.4), can further help.

Table 6.1: Bandwidth comparison (workload set #1).

Workload	Fractional solution	SLP_1	$\mathsf{Gr}^\star$	Gr
(IS:L, BI:L)	3.09E9	7.12E9	6.53E9	9.50E9
(IS:H, BI:L)	1.2E9	1.86E9	1.53E9	2.09E9
(IS:L, BI:H)	3.81E9	8.48E9	7.79E9	1.05E10
(IS:H, BI:H)	1.29E9	2.13E9	2.39E9	2.78E9

Table 6.2: Bandwidth comparison (other workload sets).

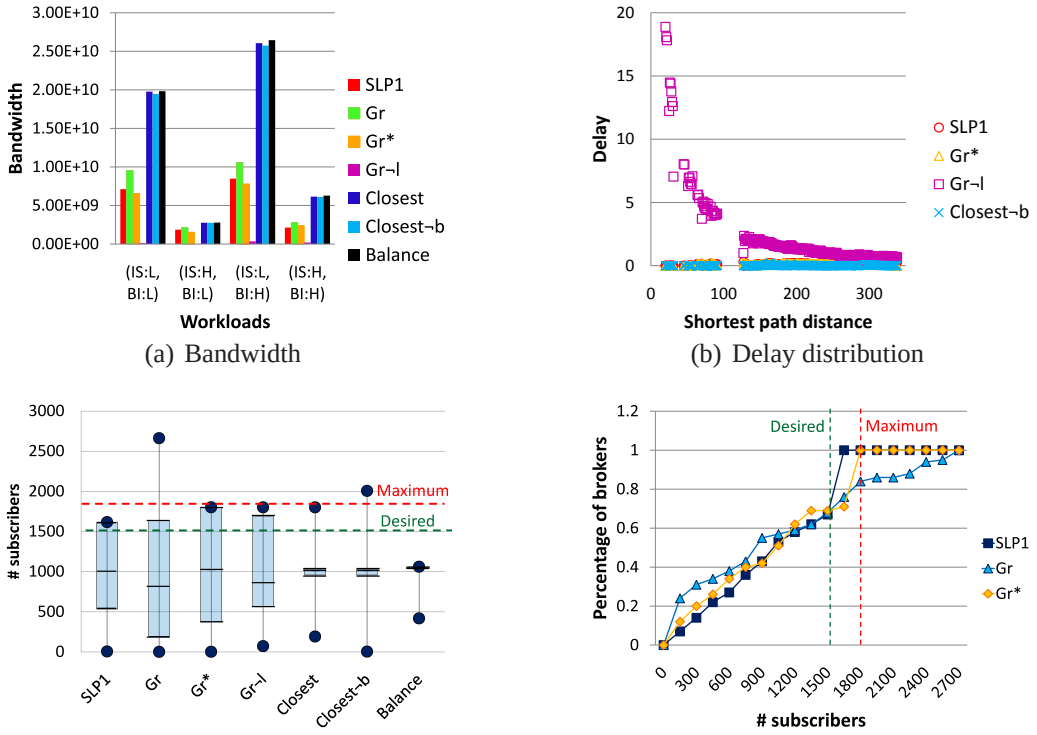
Workload set	Fractional solution	SLP_1	$\mathsf{Gr}^\star$	$\mathsf{Gr}_{\neg\mathsf{I}}$
#2	1.01E7	1.37E7	8.5E6	220
#3	2.48E10	5.4E10	5.3E10	5.09E10

Bandwidth: Figure 6.12(a), Tables 6.1 and 6.2. Figure 6.12(a) takes a closer look at total bandwidth consumption across workload set #1. The relative ordering of the algorithms is fairly consistent. SLP_1 and $\mathsf{Gr}^\star$ are good and comparable. Gr is consistently worse (not to mention its solutions also violate load balance constraints). Algorithms that ignore the event space are the worst. Again, $\mathsf{Gr}_{\neg\mathsf{I}}$ (barely visible in the figure) is just too good to be true or useful to the comparison.

Table 6.1 additionally shows the total bandwidth of the LP fractional solution obtained by running SLP_1. Recall from Section 6.4.4 that this solution provides a lower bound for the attainable bandwidth (modulo the choice of candidate filters) and the optimal bandwidth up to a small constant factor (if subscriber interests are not first clustered into super-interests). The table shows that such solutions give much more meaningful lower bounds than $\mathsf{Gr}_{\neg\mathsf{I}}$. The fact that SLP_1 and $\mathsf{Gr}^\star$ perform within small factors (between 1.3 and 2.7) from the fractional solution is a good indication that they perform very well with respect to the optimal.

Table 6.2 further shows the comparison for workload sets #2 and #3. Here, the bandwidths of the LP fractional solutions indicate that $\mathsf{Gr}^\star$ performs well in both data sets. For workload set #2, the fact that the bandwidth of $\mathsf{Gr}^\star$ is smaller than the LP fractional solution automatically implies that the bandwidth achieved by $\mathsf{Gr}^\star$ matches the lower bound (within a small constant factor).

Delays: Figure 6.12(b). Figure 6.12(b) shows scatter plots of delay versus shortest path latency for selected algorithms for (IS:H, BI:H); the results are similar for other workloads in workload set #1 and for other workload sets. Both SLP_1 and $\mathsf{Gr}^\star$ are able to bound delay at 0.3 as required.



(a) Bandwidth (b) Delay distribution

(c) Broker load distribution (d) Broker load CDF

FIGURE 6.12: Detailed comparison (one-level network, workload set #1).

Closest$_{\neg b}$ is expected to do well on delay, because it focuses exclusively on the network space. However, since Gr$_{\neg l}$ ignores the network space, it has trouble satisfying the latency constraints; subscribers near the publisher are especially vulnerable as they may be assigned to faraway brokers that blow up delays significantly.

Broker Loads: Figures 6.12(c) and 6.12(d). Figure 6.12(c) shows the boxplot of broker loads for each algorithm for (IS:H, BI:H); the results are similar for other workloads in workload set #1. The two dashed horizontal lines show the maximum and desired load bounds corresponding to $\beta_{\max}$ and $\overline{\beta}$, respectively. As expected, Balance is the best; Closest also does well because the broker distributions roughly follow the subscriber distributions in the tested workloads; Closest$_{\neg b}$ is similar to Closest but some brokers may still be overloaded because Closest$_{\neg b}$ does not enforce load balance constraints. Keep in mind, however, that these algorithms achieve good load balance at the expense of huge bandwidth (Figure 6.12(a)). Other algorithms exhibit wider range of loads.

As mentioned earlier, Gr is unable to satisfy the load balance constraints, but SLP_1, $\text{Gr}^\star$, and $\text{Gr}_{\neg\text{l}}$ do, with SLP_1 achieving a lbf close to the desired setting.

To have a closer look at the load distributions, the cumulative distribution function (CDF) is plotted for selected algorithms in Figure 6.12(d). Gr, despite its best attempt at enforcing all constraints, overloads more than 10% of the brokers.

The results are also similar for the other two workload sets. The maximum load of Gr exceeds $\beta_{\max}$ by 39% and 58% for workload sets #2 and #3, respectively.

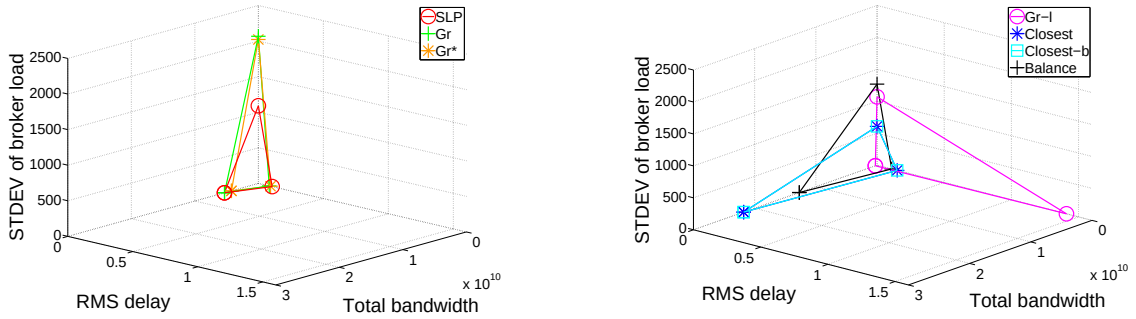
6.6.2 Solution Quality for a Multi-Level Broker Network

In the following, workload set #1 is tested and there are $100{,}000$ subscribers to assign to a multi-level network of 200 brokers, where each internal broker has a maximum out-degree of 15. The constraints are also adjusted to see how well different algorithms cope with them. In the *tight latency* setting, the maximum delay is set to 0.2; to compensate, the desired and maximum lbfs are set to 7 and 8 (the minimum possible lbf is around 6). In the *loose latency* setting, the maximum delay is set to 1, and the desired and maximum lbfs to 1.3 and 1.5.

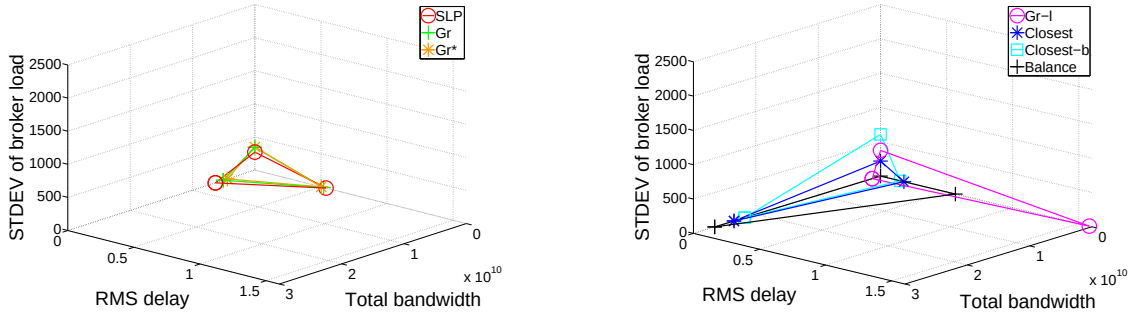
Overall Comparison: Figures 6.13(a) and 6.13(b). Similar to the results for a one-level network, algorithms that ignore the event space ($\text{Closest}_{\neg\text{b}}$, Closest, and Balance) incur high bandwidth, while the algorithm that ignores the network space ($\text{Gr}_{\neg\text{l}}$) produces long delays. Again, $\text{Gr}_{\neg\text{l}}$'s bandwidth is too unrealistic to serve as a meaningful yardstick for other solutions. Therefore, these algorithms are omitted in subsequent comparisons.

Under the loose latency setting, Gr and $\text{Gr}^\star$ are comparable to SLP, and $\text{Gr}^\star$ actually achieves slightly lower bandwidth than SLP. Under the tight latency setting, however, both Gr and $\text{Gr}^\star$ fail to produce a feasible solution that satisfies the load balance constraints (like what happened to Gr for the one-level network). Since the solution quality of $\text{Gr}^\star$ dominates that of Gr, Gr is also omitted in subsequent comparisons.

Bandwidth: Figures 6.14(a). Interestingly, for all but one of the eight workloads, SLP underperforms $\text{Gr}^\star$. One explanation is that subscribers have too few choices of brokers under the tight



(a) *tight*

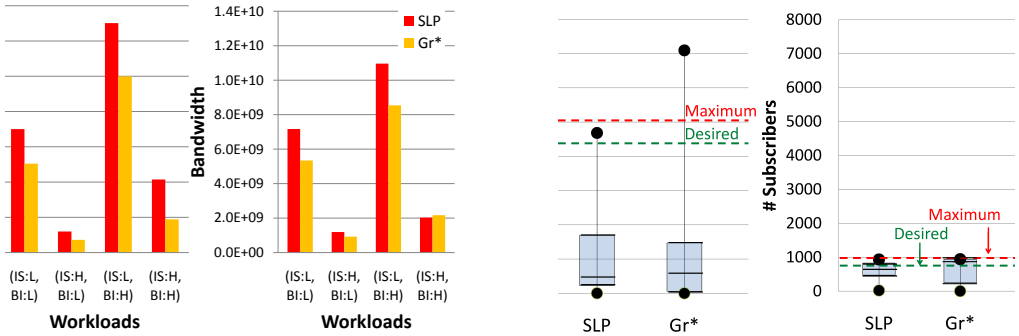


(b) *loose*

FIGURE 6.13: Overall (multi-level network, workload set #1); *tight* and *loose* refer to the tight and loose latency settings, resp.

latency setting, and too many choices under the loose setting; in either case, SLP has little advantage over Gr*. However, note that the comparison under the tight latency setting is misleading, because Gr* is unable to satisfy the load balance constraints, while SLP does. Under the loose latency setting, two algorithms actually have more similar performance.

Broker Loads: Figures 6.14(b). These figures show the results on (IS:L, BI:H). Regardless of the latency setting, SLP satisfies all constraints. On the other hand, Gr*, despite its best effort, cannot enforce all load balance constraints under the tight latency setting. A closer look at the broker load distribution (not shown here) would reveal that more than 10% of the brokers are overloaded.



(a) Bandwidth (b) Broker load

FIGURE 6.14: Other comparisons (multi-level network, workload set #1); *tight* vs. *loose*.

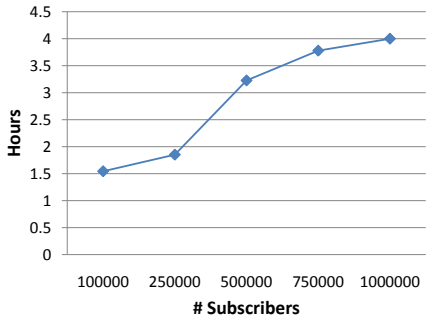


FIGURE 6.15: Running time of SLP (multi-level network).

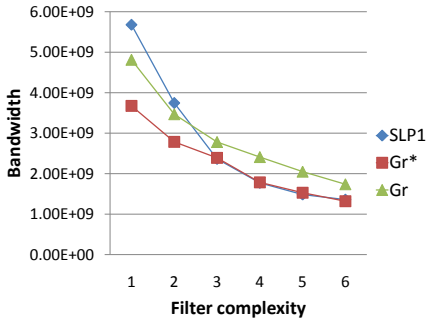
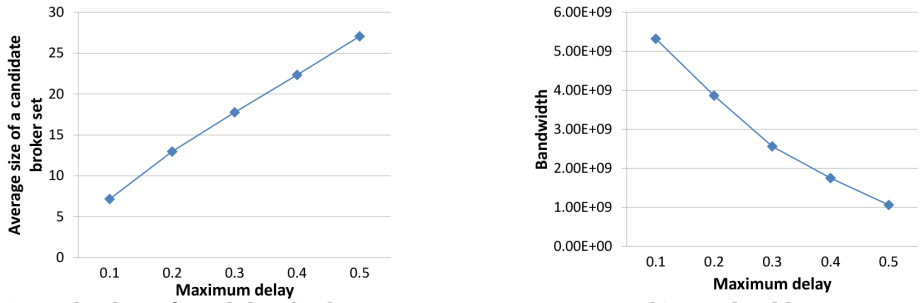


FIGURE 6.16: Effect of filter complexity (one-level network).

6.6.3 Running Time of SLP.

The wall-clock time of running SLP is measured on a Dell OptiPlex 960 desktop with Intel Core2 Duo CPU E8500 at 3.16GHz, 6144KB of cache, and 8GB of memory. The LP solver is CPLEX Version 10. A run with one million subscribers and 100 brokers in a single-level network takes about 23 hours. A run with one million subscribers and 200 brokers in a multi-level network takes about 4 hours (faster because each call to SLP_1 here involves far fewer than 100 brokers). Figure 6.6.3 shows how the number of subscribers impacts the running time of SLP.

In sum, for realistic problem sizes, SLP has manageable running time on mid-range hardware. While SLP is by no means fast, its solution quality makes it well worthwhile, especially as a yardstick to gauge other algorithms.



(a) Cardinality of candidate broker set. (b) Bandwidth.

FIGURE 6.17: Effect of maximum delay.

Table 6.3: lbf; varying broker distribution.

Broker distri.	$(44, 12, 44)$	$(66, 12, 22)$	$(22, 12, 66)$	$(33, 34, 33)$	$(47, 6, 47)$	$(22, 56, 22)$
Balance	1.062	1.674	1.674	1.062	1.974	1.062
Closest$_{\neg b}$	4.464	4.222	7.016	4.464	2.927	7.016

6.6.4 Effect of Problem Parameters.

This section considers the impact of various input parameters on the problem.

Effect of Filter Complexity. Figure 6.6.3 shows the effect of the filter complexity (α) on the total bandwidth of solutions by SLP, Gr, and Gr*. The workload is (IS:H, BI:H), with a one-level network. As discussed in Section 6.2, a larger α may reduce bandwidth, because multiple rectangles can summarize a set of interests more precisely than a single rectangle. This effect is clear and similar for all three algorithms. At the lowest α settings of 1 and 2, SLP$_1$ is more vulnerable than Gr and Gr*: a filter may consist of multiple faraway rectangles after rounding of the fractional solution; covering them with just one or two MEB may increase the filter volume dramatically. Overall, $\alpha = 3$ is a reasonable choice for all algorithms; a larger α will increase storage and processing overhead at a broker and its parent, and has diminishing effect on bandwidth.

Effect of maximum delay. Figure 6.17(a) shows how the cardinality of a candidate broker sets is affected by the parameter maximum delay. When the maximum delay is set to 0.3, each subscriber has roughly 17% brokers in its candidate broker set in average. This gives sufficient rooms for the optimization of bandwidth as shown in Figure 6.17(b).

Varying Broker Distribution in $\mathbb{N}$. Table 6.3 shows how broker distribution affects the load balance factor of the solutions. Balance aims to optimize load balancing, and Closest$_{\neg b}$ assigns subscribers to brokers without consideration of load balance. The lbf of SLP must lie in between the lbfs of Balance and Closest$_{\neg b}$. The subscriber distribution is $4 : 1 : 4$. When the broker distribution is similar to the subscriber distribution in $\mathbb{N}$, the lbf of Balance is approximately one. Its lbf becomes larger as the distributions start deviating from one another, but in the case where more brokers are located near the publisher, load balancing is always improved as subscribers have larger candidate broker sets. On the other hand, Closest$_{\neg b}$'s load imbalance can be arbitrarily bad. Hence, the maximum lbf is capped to be the lbf of Balance times 1.5, which provides rooms for SLP to minimize bandwidth while satisfying other constraints.

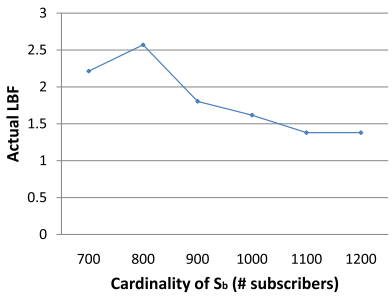
6.6.5 Algorithm Parameters.

Experiments with different choices of parameters for SLP have also been run to verify the settings of parameters.

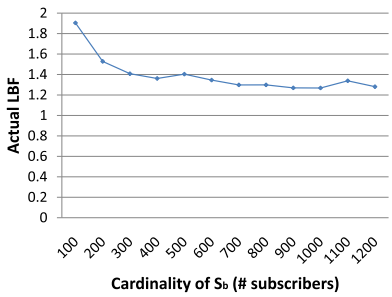
Size of $\mathcal{S}_b$ on Load Balance (Section 6.4.1.2). The load balance of the assignment depends on the number of random subscribers drawn from a uniform distribution to reflect the properties of $\mathcal{S}$ relevant to load balancing. As shown in Figure 6.18, uniformly sampling around $10|\mathcal{B}|$ subscribers is sufficient to ensure load balance ($\overline{\beta} = 1.5$ and $\beta_{\max} = 1.8$) for a one-level network. For a multi-level network, uniformly sampling around 20 times the out-degree of an internal broker returns an assignment with the load balance factor between $\overline{\beta} = 1.3$ and $\beta_{\max} = 1.5$.

Size of Ξ on Bandwidth (Section 6.4.1.3). Figure 6.19 shows how the number of super-interests affects the quality of the candidate filter set $\mathcal{R}$ by setting it to be different multiples of $|\mathcal{B}|$. As shown in the figure, when $|\Xi|$ is increased, bandwidth is gradually decreased for a one-level network, but it is only slightly improved for a multi-level network. When the out-degree of a broker is small, brokers at a higher level of the tree tend to have large filters even if the quality of the filter candidate set is further improved, and the bandwidth into those brokers dominates the bandwidth

of $\mathcal{T}$. Figure 6.20 shows the influence of the number of super-interests on the cardinality of the filter candidate set.

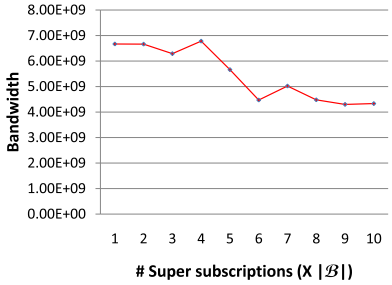


(a) One-level network ($|\mathcal{B}| = 100$)

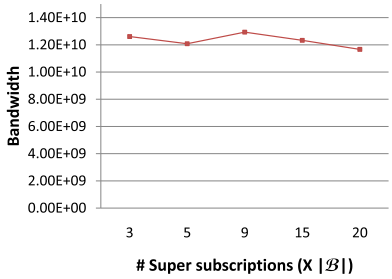


(b) Multi-level network (with out-degree 15)

FIGURE 6.18: Actual Load balance factor vs. $|\mathcal{S}_b|$.



(a) One-level network

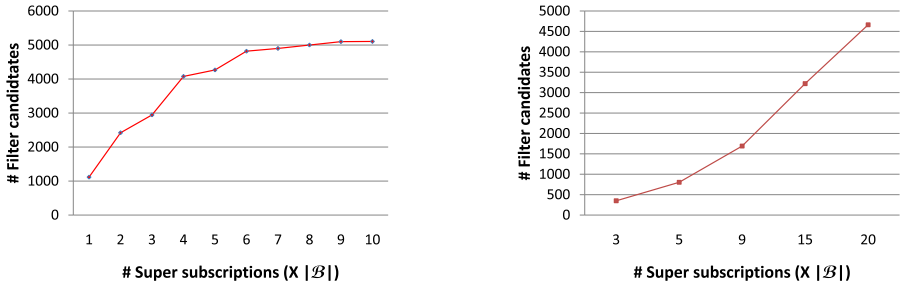


(b) Multi-level network

FIGURE 6.19: Bandwidth consumption vs. $|\Xi|$.

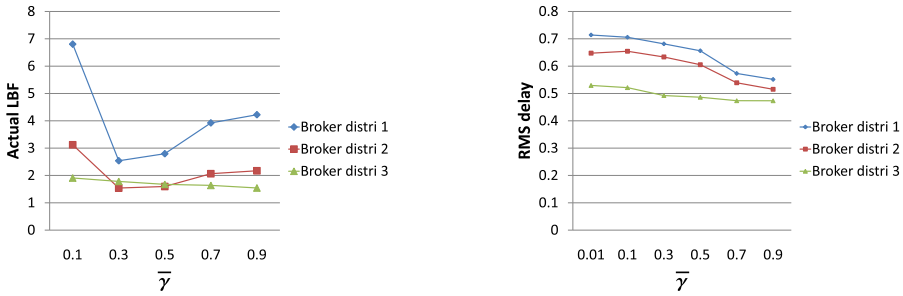
Threshold $\overline{\gamma}$ for the Multi-level Algorithm (Section 6.5). Recall that for a multi-level tree, we determine that assigning S_j to B' satisfies the latency constraint if and only if $\gamma_j(B') \geq \overline{\gamma}$. Figures 6.21 show how the threshold $\overline{\gamma}$ affects delay and load balance. The distribution of brokers across Asia, North America, and Europe is $(8:1:2)$, $(4:1:4)$, and $(2:1:8)$ for broker distributions # 1, # 2, and # 3, respectively. The subscriber distribution is $(4:1:4)$ and publisher is located in Europe.

Since the dissemination trees follow the topology of the underlying network, assigning every subscriber to a subtree with most leaf brokers satisfying its latency constriant results in smaller



(a) One-level network (b) Multi-level network

FIGURE 6.20: Cardinality of filter set vs. $|\Xi|$.



(a) Actual load balance factor vs. $\overline{\gamma}$. (b) Delay vs. $\overline{\gamma}$

FIGURE 6.21: Threshold $\overline{\gamma}$ for the multi-level algorithm.

latency from the publisher to the subscriber. As expected, both low and high thresholds disallow subscribers to be distributed evenly and the actual load balance factor is bad for both cases.

6.6.6 *Discussion.*

One take-way point from these experiments is that Gr* works well on many (though not all) workloads, including fairly realistic ones generated from statistics on Google Groups. What is more important, however, is what allows us to draw this conclusion. Solutions obtained by algorithms that ignore any performance criterion are not helpful—not only do they tend to fare terribly on criteria they ignore, but they also cannot offer meaningful bounds on what can be realistically achieved. On the other hand, the LP-based approach is a better yardstick for evaluating different algorithms. While we cannot guarantee the optimality of SLP$_1$, we have more assurance of its solution quality (Section 6.4.4) across problem instances. Furthermore, the fractional solution it produces gives us

another indicator of optimality that is far more useful than, say, what $\mathsf{Gr}_{\neg|}$ offers.

One might wonder if $\mathsf{Gr}^\star$ works well in general. It does not. We have already seen that it has trouble with load balance constraints under the tight latency setting. Furthermore, the next section will show how to construct concrete problem instance for which $\mathsf{Gr}^\star$ performs orders of magnitude worse than SLP. This example further illustrates the importance of developing better yardsticks for evaluating algorithms for SA.

6.6.7 A Difficult Workload for $\mathsf{Gr}^\star$.

$\mathsf{Gr}^\star$ works well for most cases studied, but a counterexample can be constructed easily. $\mathsf{Gr}^\star$ performs poorly on the counterexample because it is forced to make a costly assignment for subscribers appeared late in the assignment sequence. Although $\mathsf{Gr}^\star$ defers the processing of subscribers with more choices, the choices available to those subscribers can become limited because most brokers become fully loaded or simply because of tight latency, in which case all subscribers have few choices. However, $\mathsf{Gr}^\star$ is expected to perform well as long as the capacity and latency constraints are not too tight.

Given filter complexity α, the idea is to construct a sorted sequence of subscribers such that $\mathsf{Gr}^\star$ will assign $\alpha + 1$ well separated rectangles to each broker; merging any pair of the rectangles will create a large rectangle. The workload of m subscribers and n brokers is constructed as follow. $(\alpha+1)n$ interests are created, each of which is a unit square centered at a point on the line $y = x$ in $\mathbb{E} = \mathbb{R}^2$. Each interest has $\overline{\beta}m/n$ subscribers. Let $I_1, I_2, \cdots, I_{(\alpha+1)n}$ be the sequence of interests in ascending order of the x-axis. For all $i < (\alpha + 1)n$, let the distance between the centers of interests I_i and I_{i+1} be $10^{(i \bmod \alpha)+1}\sqrt{2}$. An example of interests for $\alpha = 3$ and $n = 3$ is shown in Figure 6.22. Next, for each subscriber S_j, we define a subset of brokers to which S_j can be assigned without violating latency constraints. Every subscriber S_j that has interest in I_i can be assigned to any broker if $i > \alpha n$, otherwise, S_j can only be assigned to $\mathcal{B}_j =$

$$\begin{cases} \{B_{\lfloor i/(\alpha+1)\rfloor+1}, B_{(i \bmod n)+1}\} & \text{if } \lfloor i/(\alpha+1)\rfloor \not\equiv i \bmod n, \\ \{B_{\lfloor i/(\alpha+1)\rfloor+1}, B_{(i \bmod n)}\} & \text{otherwise.} \end{cases}$$

An example of feasible broker sets for $\alpha = 3$ and $n = 3$ is shown in Table 6.4.

Let $I_i \rightarrow I_j$ denote that all the subscribers who are interested in I_i are in front of those who are interested in I_j in the sequence. Subscribers are initially sorted in ascending order of the cardinality of their candidate broker sets: $I_1 \rightarrow I_2 \rightarrow \cdots \rightarrow I_{\alpha n-1} \rightarrow I_{\alpha n} \rightarrow I_{(\alpha+1)n} \rightarrow I_{(\alpha+1)n-1} \rightarrow \cdots \rightarrow I_{\alpha n+2} \rightarrow I_{\alpha n+1}$. Though the ordering of the remaining subscribers is always updated, Gr* attaches the subscribers with interests $I_{i+j-1+kn}$ to the same broker B_j, where $j \in \{1, 2, \cdots, n\}$ and $k \in \{0, 1, \cdots, \alpha\}$. The subscriber assignment for $\alpha = 3$ and $n = 3$ is shown in Figure 6.24. In order to satisfy the filter complexity, two interests are forced to be covered by the same huge rectangle.[5] The bandwith consumption is roughly $3 * 10^6$, which is 10^4 times worse than the result of SLP. As shown in Figure 6.23, SLP minimizes bandwidth consumption by attaching the subscribers with interests I_{ij}, I_{ij+1}, I_{ij+2}, and I_{ij+3} to the same broker B_j, for $j \in \{1, 2, 3\}$. The cost is roughly 300. In fact, this is the optimal solution.

Table 6.4: An example for $\alpha = 3$ and $n = 3$.

Subscribers interested in	Initial set of candidate brokers
I_1, I_2, I_4, I_5, I_7, or I_8	$\{B_1, B_2\}$
I_3 or I_9	$\{B_1, B_3\}$
I_6	$\{B_2, B_3\}$
I_{10}, I_{11}, or I_{12}	$\{B_1, B_2, B_3\}$

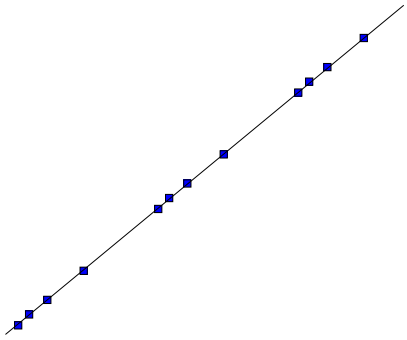


FIGURE 6.22: Interests in $\mathbb{E}$ with $\alpha = 3$ and $n = 3$.

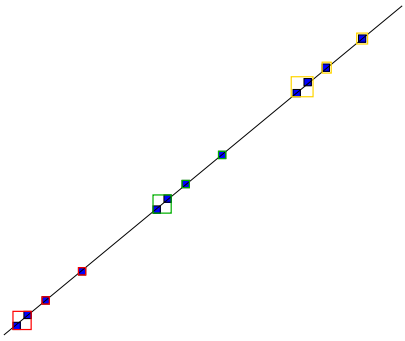
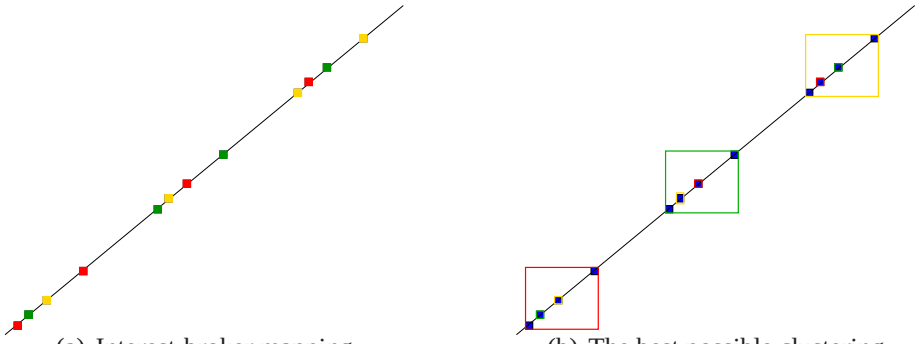


FIGURE 6.23: Filters generated by SLP.

[5] As the gap between the values of $\overline{\beta}$ and $\beta_{\max}$ is increased, some filters will not have a huge rectangle because subscribers for the 4^{th} interest may be assigned to other brokers. However, one can increase data skewness (ex: increase the number of subscribers for the 4^{th} interest) such that the performance of Gr* remains orders of magnitude worse than that of SLP.



(a) Interest-broker mapping (b) The best possible clustering

FIGURE 6.24: Filters generated by Gr*. Interests with the same color are handled by the same broker.

6.7 Related work

Dissemination network design for publish/subscribe has received much attention in the past few years. As discussed in Section 6.1, some previous work considers either interest similarity in the event space (e.g., [53]) or subscriber location in the network space [13] while ignoring the other aspect. Other performance objectives and constraints have also be considered in subscriber assignment. Shah et al. [104] maximize data fidelity. Tariq et al. [112] maximize the number of subscribers whose latency constraints are satisfied without violating bandwidth constraints.

Another line of research focuses on self-organizing, distributed algorithms that dynamically reconfigure the network topology to optimize specific measures. Baldoni et al. [21] minimize the number of hops and let subscribers be uniformly spread among brokers. Jaeger et al. [71] minimize total processing and communication costs (excluding last-hop latencies between brokers and subscribers). The distribution of subscribers to brokers is chosen probabilistically according to a random load value. Papaemmanouil et al. [91] present a general optimization framework that iteratively improves performance, starting by randomly attaching subscribers to a node. Understanding the robustness and global optimality of such algorithms has been challenging. The work presented in this chapter complements this line of research by offering a yardstick for evaluation that is computationally feasible over more realistic problem sizes.

Distributed stream processing is also related to the work presented in this chapter. Stream processing systems process and aggregate data over a network of machines, and one key issue is how

to optimally place query operators onto the set of machines (see [75] for overview and [123, 92] for more recent development). However, the number of queries involved in the operator placement problem is orders of magnitude smaller than the number of subscribers in the subscriber assignment problem.

There is a vast body of literature on network design in general. The minimum steiner tree problem [25] and the weighted steiner tree packing problem [60] resemble the publish/subscribe network overlay construction problem if steiner points are viewed as brokers, and terminals are viewed as publishers and subscribers. The minimum steiner tree problem is APX-hard [25]. Migliavacca and Cugola [84] have studied the optimal content-based routing problem, which is to find a minimal subtree that connects all subscribers who share the same interest, such that the total communication and processing costs are minimized.

Finally, researchers have studied network design in the area of content distribution networks [54, 22]. While a content distribution network is not a pure dissemination system but more of a hybrid between push and client pull, it faces similar issues such as balancing load and bounding latency.

6.8 Conclusion and Future Work

This chapter has presented SLP, a LP-based algorithm for SA, the subscriber assignment problem for wide-area content-based publish/subscribe. SLP considers the subscriber distribution in both event and network spaces to minimize bandwidth while satisfying latency and load balance constraints. To ensure its scalability to realistic problem sizes, SLP employs a suite of techniques, including LP relaxation, randomized rounding, coreset, sampling, and max-flow, to carefully reduce its complexity.

As a solution to the offline SA problem, SLP can be used for initial subscriber assignment and periodical re-optimization. More importantly, with better theoretical properties and robustness to workload variations, SLP serves as a reasonable yardstick for evaluating simpler heuristic algorithms across realistic workloads in both online and offline settings. Using this yardstick, it is shown that an efficient greedy algorithm, Gr*, works well for a number of workloads. Compared with previous work, this chapter has pushed the sophistication and scale of evaluation to new

heights.

There are two immediate directions for future work. First, a principled approach is still much needed for the dynamic version of the subscriber assignment problem, where subscribers come and go. Second, it would be good to drop the assumption that a broker tree is given in advance, and jointly optimize subscriber assignment, broker placement, as well as the dissemination network topology.

6.9 Theorems and Proofs

Proof of Lemma 28. For the sake of readability, we bound the size of an ϵ-certificate for $d = 2$ and the maximum filter complexity equal to one. The proof can be extended to arbitrary dimensions and arbitrary filter complexities analogously.

If there is only one single broker, there are two cases: (1) If the broker cannot satisfy all user-specified latency constraints, no certificate exists and $\emptyset$ is returned; (2) otherwise, an ϵ-certificate consists of subscribers whose interests are the leftmost, rightmost, up-most, and bottom-most in $\mathbb{E}$.

For $|\mathcal{B}| > 1$, we pick an arbitrary subscriber $S_j \in \mathcal{S}$. Let its interest s_j be $[\ell_1, h_1] \times [\ell_2, h_2]$. We place an exponential grid centered at $(\frac{\ell_1+h_1}{2}, \frac{\ell_2+h_2}{2})$. Let $w_{i,\beta,\alpha} = (h_i - \ell_i)(2^\beta(1+\alpha\epsilon/2)-1)$. The grid consists of vertical lines $\{x = \ell_1 - w_{1,\beta,\alpha}, x = h_1 + w_{1,\beta,\alpha}\}$ and horizontal lines $\{y = \ell_2 - w_{2,\beta,\alpha}, y = h_2 + w_{2,\beta,\alpha}\}$, where $\alpha \in [1, 2, 3, \cdots, \lfloor 2/\epsilon \rfloor]$ and $\beta \in [0, 1, 2, \cdots, \log_2 \Delta]$, as shown in Figure 6.25. Let $\mathcal{R}_j$ be the set of rectangles whose lower-left corners are (brown) grid points in the southwest quadrant of point (ℓ_1, ℓ_2) and whose upper-right corners are (blue) grid points in the northeast quadrant of point (h_1, h_2). Let $\mathcal{B}_j$ be the subset of brokers that satisfy the user-specified latency constraint for S_j if S_j is assigned to them. For each $B_i \in \mathcal{B}_j$ and each rectangle $R \in \mathcal{R}_j$, let $\mathcal{S}_i^R$ be the set of subscribers that are not covered by B_i if filter $f_i = R$; we find an ϵ-certificate $\mathcal{Q}_i^R$ for $\mathcal{B}\backslash\{B_i\}$ and $\mathcal{S}\backslash\{\mathcal{S}_i^R\}$. An ϵ-certificate for $\mathcal{B}$ and $\mathcal{S}$ is:

$$\mathcal{Q} = \bigcup_{R\in\mathcal{R}_j, B_i\in\mathcal{B}_j} \mathcal{Q}_i^R.$$

Without loss of generality, say S_j is assigned to B_i. Let $R \in \mathcal{R}_j$ be the smallest rectangle containing filter f_i. By construction, an ϵ-expansion of f_i would contain R, so every subscriber in $\mathcal{S}_i^R$ is covered by $(1+\epsilon)f_i$. Since $\mathcal{Q}$ also includes an ϵ-certificate for $\mathcal{B}\backslash\{B_i\}$ and $\mathcal{S}\backslash\{\mathcal{S}_i^R\}$, $\mathcal{Q}$ is an

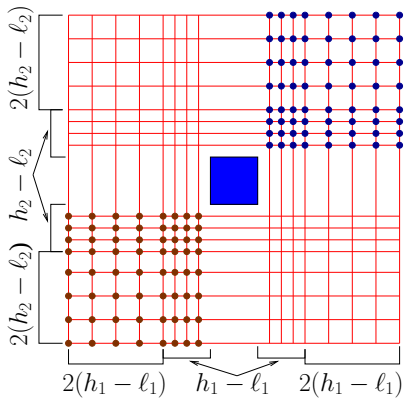


FIGURE 6.25: Two levels ($\beta = \{0, 1\}$) of exponential grid with ϵ set to $1/2$.

ϵ-certificate for $\mathcal{S}$ and $\mathcal{B}$.

The cardinalities of $\mathcal{R}_j$ and $\mathcal{B}_j$ are $O((\log_2 \Delta/\epsilon)^4)$ and $O(n)$, resp. Since one broker is removed from $\mathcal{B}_j$ for each $\mathcal{Q}_i^R$, the size of an ϵ-certificate is easily verified to be $O((n(\log_2 \Delta/\epsilon))^{4n}$ by solving the recursive function $g(n) = n(\log_2 \Delta/\epsilon)^4 g(n-1)$. □

Proof of Lemma 29. The lemma directly follows from the fact that for each dimension, an interval of length between $\ell_j/4$ and $\ell_j/2$ is contained by at least one interval in $\mathcal{J}_{ij}$. □

Lemma 30 (Number of iterations)**.** *If no certificate is found after $4g \log_2(|\mathcal{S}|/g)$ iterations, the size of a certificate must be greater than g.*

Proof. The analysis is similar to [46, 30]. Let $w(\mathcal{X})$, where $\mathcal{X}$ is a set of subscribers, be a shorthand for $\sum_{S \in \mathcal{X}} w(S)$. Let $\mathcal{Q}$ be a certificate with g subscriber interests and suppose we have not found any coreset after l iterations. For every round, there must be at least one interest in $\mathcal{Q}$ that is not covered by the ε-expansion of Φ (otherwise, by covering $\mathcal{Q}$, we would have found a certificate), and its weight is doubled. Hence, $w(\mathcal{Q}) \geq g \cdot 2^{l/g}$ after l iterations. On the other hand, the validity condition (Line 14 of Algorithm 9) ensures that the total weight of the interests not covered by the ε-expansion of Φ is always at most $w(\mathcal{S})/8$, so doubling the weights of those interests cannot increase $w(\mathcal{S})$ by more than a factor of $(1 + 1/8)$. Therefore, $w(\mathcal{S}) \leq |\mathcal{S}|(1 + 1/8)^l$ after l iterations. From $g \cdot 2^{l/g} \leq w(\mathcal{Q}) \leq w(\mathcal{S}) \leq |\mathcal{S}|(1+1/8)^l < |\mathcal{S}|e^{l/(2g)} < |\mathcal{S}| \cdot 2^{3l/(4g)}$, we conclude that $l < 4g \log_2(|\mathcal{S}|/g)$. □

Lemma 31 (Probability of valid round)**.** *Let $\mathcal{Q}$ be a random sample of size $cg \ln g$, where c is a constant, and Φ be the set of filters assigned to $\mathcal{B}$ to cover $\mathcal{Q}$. Let $\mathcal{S}' \subseteq \mathcal{S}$ be a set of subscribers not covered by Φ. The probability that $W(\mathcal{S}') > \epsilon W(\mathcal{S})$ is at most 1/2.*

Proof. Recall that subscriber S_j can be assigned to broker B_i only if 1) its interest s_j is contained by filter f_i in $\mathbb{E}$, and 2) the network coordinate of S_j is within $\delta_j - \lambda_i$ units away from that of B_i in $\mathbb{N}$, where δ_j is the maximum allowable latency for S_j and λ_i is the path latency from the publisher to broker B_i in $\mathcal{T}$. Consider the L_∞ norm. In $\mathbb{N}$, let φ_j be a rectangle of width $2\delta_j$ centered at S_j and ϱ_i be a rectangle of width $2\lambda_i$ centered at B_i. The second condition is equivalent to "ϱ_i is contained by φ_j in $\mathbb{N}$."

Let $\mathbb{X} = \mathbb{R}^{d+t}$ be the combined space of $\mathbb{E}$ and $\mathbb{N}$. For simplicity, each of the n brokers has a rectangle filter. The argument can be extended for higher filter complexity. Let $\Sigma^n(\mathcal{S}, R^n)$ be a range space, where a range $X \in R^n$ is defined as the compliment of the union of n rectangles in $\mathbb{X}$. Since the range is defined by combinations of $4(d+t)n$ linear inequalities, $\mathsf{VC\text{-}dim}(\Sigma^n) = O((d+t)^2 n \ln((d+t)n))$. Since the VC-dimension of the range space is finite, the lemma follows from the theory of ϵ-nets [64] by choosing the constant c larger than the VC dimension, which depends on d, t, and n. □

Proof of Theorem 27. The proof consists of four components: bandwidth, filter complexity, latency and nesting, and load balance:

(i) [Bandwidth] $\mathsf{E}[\sum_{B_i \in \mathcal{B}, R_k \in \mathcal{R}} \mathrm{Vol}(R_k) y_{ik}] =$

$\sum_{B_i \in \mathcal{B}, R_k \in \mathcal{R}} \mathrm{Vol}(R_k) \mathsf{E}[y_{ik}] \leq \sum_{B_i \in \mathcal{B}, R_k \in \mathcal{R}} \mathrm{Vol}(R_k) \ln |\mathcal{S}_a| \hat{y}_{ik} = (\ln |\mathcal{S}_a|) \mathrm{OPT}_{\mathrm{LP}}$.

(ii) [Filter complexity] $\mathsf{E}[\sum_{R_k \in \mathcal{R}} y_{ik}] = \sum_{R_k \in \mathcal{R}} \mathsf{E}[y_{ik}] = \sum_{R_k \in \mathcal{R}} (\ln |\mathcal{S}_a|) y_{ik} \leq (\ln |\mathcal{S}_a|)\alpha$.

(iii) [latency and nesting] Here, we show that there exists a rounding scheme for variables x_{ij}, such that the latency and nesting constraints can be enforced with probability at least $1/e$. We round variables x_{ij}'s as follows:

$$\Pr[x_{ij} = 1 \mid y] = \begin{cases} \frac{1 - |\mathcal{S}_a|^{-\hat{x}_{ij}}}{1 - \prod_{R_k \in \mathcal{R}_j} (1 - \hat{y}_{ik})^{\ln |\mathcal{S}_a|}} & \text{if } \sum_{R_k \in \mathcal{R}_j} y_{ik} \geq 1, \\ 0 & \text{otherwise.} \end{cases}$$

This ensures that a subscriber S_j is assigned to broker B_i only if B_i covers S_j. Also, $\Pr[x_{ij} = 1 \mid y]$ is always between 0 and 1 since constraints (C4) ensures that $\sum_{R_k \in \mathcal{R}_j} y_{ik} \geq x_{ij}$, which implies $1 - |\mathcal{S}_a|^{-\hat{x}_{ij}} \leq 1 - |\mathcal{S}_a|^{-\sum_{R_k \in \mathcal{R}_j} y_{ik}} \leq 1 - \prod_{R_k \in \mathcal{R}_j} (1 - y_{ik})^{\ln |\mathcal{S}_a|}$.

Recall that $\Pr[y_{ik} = 1] = 1 - (1 - \hat{y}_{ik})^{\ln |\mathcal{S}_a|}$. The probability that broker B_i covers S_j is $\Pr[\sum_{R_k \in \mathcal{R}_j} y_{ik} \geq 1]$ = $1 - \Pr[\sum_{R_k \in \mathcal{R}_j} y_{ik} = 0]$ = $1 - \prod_{R_k \in \mathcal{R}_j} (1 - \hat{y}_{ik})^{\ln |\mathcal{S}_a|}$. The probability that subscriber S_j is assigned to broker B_i is equal to the sum of $\Pr[x_{ij} = 1 \mid \sum_{R_k \in \mathcal{R}_j} y_{ik} \geq 1] \cdot \Pr[\sum_{R_k \in \mathcal{R}_j} y_{ik} \geq 1]$ and $\Pr[x_{ij} = 1 \mid \sum_{R_k \in \mathcal{R}_j} y_{ik} = 0] \cdot \Pr[\sum_{R_k \in \mathcal{R}_j} y_{ik} = 0]$. A straight forward calculation will give $\Pr[x_{ij} = 1] = 1 - |\mathcal{S}_a|^{-\hat{x}_{ij}}$.

The probability that a subscriber S_j is not assigned to any broker is $\Pr[\cap_{B_i \in \mathcal{B}_j} \{x_{ij} = 0\}] = \prod_{B_i \in \mathcal{B}_j} \Pr[\{x_{ij} = 0\}] = \prod_{B_i \in \mathcal{B}_j} |\mathcal{S}_a|^{-\hat{x}_{ij}} = |\mathcal{S}_a|^{\sum_{B_i \in \mathcal{B}_j} -\hat{x}_{ij}} \leq |\mathcal{S}_a|^{-1}$. Hence, the probability of every subscriber assigned to a broker is at least $\prod_{S_j \in \mathcal{S}_a} \Pr[\bigcup_{B_i \in \mathcal{B}_j} \{x_{ij} = 1\}] = \prod_{S_j \in \mathcal{S}_a} (1 - \Pr[\bigcap_{B_i \in \mathcal{B}_j} \{x_{ij} = 0\}]) \geq \prod_{S_j \in \mathcal{S}_a} (1 - |\mathcal{S}_a|^{-1}) = (1 - |\mathcal{S}_a|^{-1})^{|\mathcal{S}_a|} \geq 1/e$.

(iv) [Load balance] Using the above rounding scheme, $\mathsf{E}[\sum_{S_j \in \mathcal{S}_b} x_{ij}] = \sum_{S_j \in \mathcal{S}_b} \mathsf{E}[x_{ij}]$ $= \sum_{S_j \in \mathcal{S}_b} (\ln |\mathcal{S}_a|) x_{ij} \leq (\ln |\mathcal{S}_a|) \overline{\beta} \kappa_i |\mathcal{S}_b|$. □

7

Conclusion and Future Work

7.1 Conclusion

This dissertation has examined the problem of answering various types of user queries in various settings. The problem was modeled using a geometric framework. By applying techniques such as dual transform, coreset, sampling and dimensionality reduction, efficient algorithms were developed for both query processing and notification dissemination.

Chapter 3 addressed the problem of supporting a large number of continuous preference top-k queries. The chapter proposed the notion of QRS (query response surface) and developed solutions within a geometric framework. Recognizing the connection to halfspace range queries, data structures were obtained for reverse top-k queries with linear space and sublinear query time. Building on this result, a fully dynamic solution was presented to scale the solution to a million preferences with both object and preference updates. The presented duality-based approach enabled effective subscription clustering; for regions where subscriptions were heavily clustered, queries were jointly processed to acheive better performance. This chapter also defined and solved the approximate preference top-k queries. The presented coreset-based approach significantly improved the query time of the algorithms with only little loss in accuracy. Experimental evaluation confirmed the effectiveness of our ideas such as selective QRS-driven processing and coreset-based QRS

simplification, which helped advance our solutions in both scalability and functionality.

Chapter 4 presented efficient data structures and algorithms for top-k and reverse top-k queries in high dimensions. Our solution was based on the idea of core subspaces: It exploited the sparsity in preferences to identify core subspaces, and applied the techniques of coresets and duality transform to index each core subspace as well as the full-dimensional space effectively. Experimental evaluation showed that in high dimensions, exact methods were slow, while existing approximation methods suffered from either poor speed (e.g., when using only a single coreset in the full space) or poor accuracy (such as the PCA- and view-based approaches). In contrast, for workloads where preferences were often sparse, our solution offered a desirable trade-off between speed and accuracy, which made scalable processing of preference top-k and reverse top-k queries in high dimensions a reality.

Chapter 5 tackled the problem of supporting a large number of range top-k subscriptions in a wide-area network. The chapter addressed the dual challenges of subscription processing and notification dissemination, by carefully separating and interfacing these tasks in a way that achieved efficiency with off-the-shelf dissemination networks and without increasing system complexity. Our techniques were based on a geometric framework, enabling us to characterize the subset of subscriptions affected by an event as a region in an appropriate defined space, and solved the problem of notifying affected subscriptions as one of tiling the region with basic geometric shapes. The array of techniques this chapter had developed—ranging from those that used the knowledge of subscriptions to those that did not, from event-at-time to batch processing, from exact to approximate, and from one-dimensional to multi-dimensional ranges—spoke to the power of this framework. Theoretical analysis and empirical evaluation showed that our approach and techniques held substantial advantages over less sophisticated ones.

Chapter 6 studied how to design an efficient dissemination network for range queries. In particular, the subscriber assignment (SA) problem was solved for wide-area content-based publish/subscribe. This chapter presented a LP-based algorithm called SLP, which considered the subscriber distribution in both event and network spaces to minimize bandwidth while satisfying latency and load balance constraints. To ensure its scalability to realistic problem sizes, SLP em-

ployed a suite of techniques, including LP relaxation, randomized rounding, coreset, sampling, and max-flow, to carefully reduce its complexity. As a solution to the offline SA problem, SLP could be used for initial subscriber assignment and periodical reoptimization. More importantly, because of its better theoretical properties and robustness to workload variations, SLP served as a reasonable yardstick for evaluating simpler heuristic algorithms across realistic workloads in both online and offline settings. Using this yardstick, this chapter concluded that a simple and efficient greedy algorithm, *Gr**, worked well for a number of workloads.

7.2 Future work

This section provides several directions for future research that extend the work of this dissertation:

Top-k subscriptions. In addition to object ranking, users can also be ranked from publishers' perspective: instead of disseminating updates to all affected users, a publisher will find t most relevant users to the update and only notify those t users of the updates. Recently, Sadoghi and Jacobsen [102, 103] presented data structures for supporting top-t matching subscriptions, where each subscription specifies both range condition and customized scoring function. Their solution is based on a two-phase space-cutting technique—*space partitioning* which chooses the best splitting attribute and *space clustering* which clusters users based on their subscription ranges in the splitting attribute. However, supporting top-t matching subscriptions remains an open problem when each subscription is a continuous preference top-k range query.

Continuous top-k queries under uncertainty. Another future direction is to explore different ways of modeling user interests in order to handle uncertainty in preference vectors. Users may not explicitly know the combination of weights that reflect their interests. One possibility is to model a user preference as a set of possible vectors or a cone instead of one "precise" vector.

Spatial-temporal top-k preferences Another future direction is to consider both space and time in ranking. User preferences may be location-based; users may only be interested in updates within

their local neighborhood. For example, Nextdoor,[1] a private social network, maintains thousands of neighborhoods and supports notification dissemination of local updates, such as break-in and missing pets. Section 5.5 has generalized the framework to handle k-nearest-neighbor queries, but still the ranking of objects may depend on their publication time as well; their rankings may drop over time gradually. For example, a news story about a robbery that occurred today may rank higher than another news story about a robbery that occurred a week ago. As another example, Facebook uses EdgeRank, which takes time decay into consideration, to rank the stories in the news feed.

[1] `http://nextdoor.com`

Bibliography

[1] I. Aekaterinidis and P. Triantafillou. Content-based publish/subscribe systems over structured p2p networks. In *Proc. of the 2004 Intl. Workshop on Distributed Event Based Systems*, May 2004.

[2] P. Afshani and T. M. Chan. Optimal halfspace range reporting in three dimensions. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 180–186, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[3] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *J. Comput. Syst. Sci.*, 61(2):194–216, 2000.

[4] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives, 1999.

[5] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM*, 51:606–635, 2004.

[6] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Geometric approximation via coresets. In J. E. Goodman, J. Pach, and E. Welzl, editors, *Combinatorial and Computational Geometry*, pages 1–30. Cambridge University Press, New York, 2005.

[7] P. K. Agarwal, S. Har-Peled, and H. Yu. Robust shape fitting via peeling and grating coresets. *Discrete & Computational Geometry*, 39(1-3):38–58, 2008.

[8] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, pages 325–345, 1995.

[9] P. K. Agarwal, J. M. Phillips, and H. Yu. Stability of epsilon-kernels. In *Proceedings of the 18th annual European conference on Algorithms: Part I*, ESA'10, pages 487–499, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] P. K. Agarwal, C. M. Procopiuc, and K. R. Varadarajan. Approximation algorithms for k-line center. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 54–63, London, UK, 2002. Springer-Verlag.

[11] P. K. Agarwal and M. Sharir. Arrangements and their applications. In *Handbook of Computational Geometry*, pages 49–119. Elsevier Science Publishers B.V. North-Holland, 1998.

[12] P. K. Agarwal and S. Suri. Surface approximation and geometric partitions. *SIAM J. Comput.*, 27:1016–1035, August 1998.

[13] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 1999 ACM Symposium on Principles of Distributed Computing*, pages 53–61, Atlanta, Georgia, USA, May 1999.

[14] Akamai and JupiterResearch. Identify 4 seconds as new threshold of acceptability for retail web page response times, 11 2006. Press Release, Akamai Technologies Inc, `www.akamai.com/html/about/press/releases/2006/press_110606.html`.

[15] A. Aldroubi. A review of subspace segmentation: Problem, nonlinear approximations, and applications. *Signal Processing Review*, 2012.

[16] M. Altmel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 2000 International Conference on Very Large Data Bases*, Cairo, Egypt, Sept. 2000.

[17] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[18] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[19] A. Asuncion and D. Newman. UCI machine learning repository, 2007.

[20] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 28–39, New York, NY, USA, 2003. ACM.

[21] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *The Computer Journal*, 7 2007.

[22] N. Ball and P. Pietzuch. Distributed content delivery using load-aware network coordinates. In *CoNEXT '08: Proceedings of the 2008 ACM CoNEXT Conference*, pages 1–6, New York, NY, USA, 2008. ACM.

[23] G. Banavar, T. Ghandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 262, Washington, DC, USA, 1999. IEEE Computer Society.

[24] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '99, pages 82–91, Philadelphia, PA, USA, 1999. Society for Industrial and Applied Mathematics.

[25] M. Bern and P. Plassmann. The steiner problem with edge lengths 1 and 2,. *Inf. Process. Lett.*, 32(4):171–176, 1989.

[26] S. Bianchi, P. Felber, and M. Gradinariu. Content-based publish/subscribe using distributed r-trees. *Euro-Par*, 4641:537–548, 2007.

[27] V. Bilò, I. Caragiannis, C. Kaklamanis, and P. Kanellopoulos. Geometric clustering to minimize the sum of cluster sizes. In *In Proc. 13th European Symp. Algorithms, Vol 3669 of LNCS*, pages 460–471, 2005.

[28] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, Washington, DC, USA, 2001. IEEE Computer Society.

[29] H. Brönnimann, B. Chazelle, and J. Pach. How hard is half-space range searching. *Discrete & Computational Geometry*, 10:143–155, 1993.

[30] H. Brönnimann and M. T. Goodrich. Almost optimal set covers in finite vc-dimension. *Discrete and Computational Geometry*, 14:463–479, 1995.

[31] C. Buchta. On the average number of maxima in a set of vectors. *Inf. Process. Lett.*, 33:63–65, November 1989.

[32] R. Buyya, M. Pathan, and A. Vakali. *Content Delivery Networks*. Springer, Berlin, Germany, 2008.

[33] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 206–215, New York, NY, USA, 2004. ACM.

[34] A. Carzaniga, D. S. Rosenblum, , and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), 2001.

[35] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *IMWS '01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, pages 59–68, London, UK, 2002. Springer-Verlag.

[36] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Journal on Selected Areas in Communication*, 20, October 2002.

[37] I. Ceaparu, J. Lazar, K. Bessière, J. P. Robinson, and B. Shneiderman. Determining causes and severity of end-user frustration. *Int. J. Hum. Comput. Interaction*, 17(3):333–356, 2004.

[38] T. M. Chan. Three problems about dynamic convex hulls. In *Proceedings of the 27th annual ACM symposium on Computational geometry*, SoCG '11, pages 27–36, New York, NY, USA, 2011. ACM.

[39] R. Chand and P. A. Felber. A scalable protocol for content-based routing in overlay networks. In *NCA '03: Proceedings of the Second IEEE International Symposium on Network Computing and Applications*, page 123, Washington, DC, USA, 2003. IEEE Computer Society.

[40] B. Chandramouli, J. M. Phillips, and J. Yang. Value-based notification conditions in large-scale publish/subscribe systems. In *Proceedings of the 2007 International Conference on Very Large Data Bases*, pages 878–889, Vienna, Austria, Sept. 2007.

[41] B. Chandramouli, J. Xie, and J. Yang. On the database/network interface in large-scale publish/subscribe systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 587–598, Chicago, Illinois, USA, June 2006.

[42] B. Chandramouli and J. Yang. End-to-end support for joins in large-scale publish/subscribe systems. In *Proceedings of the 2008 International Conference on Very Large Data Bases*, pages 434–450, Auckland, New Zealand, Aug. 2008.

[43] H. Chang, R. Govindan, S. Jamin, S. J. Shenker, and W. Willinger. Towards capturing representative as-level internet topologies. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '02, pages 280–281, New York, NY, USA, 2002. ACM.

[44] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. *SIGMOD Rec.*, 29:391–402, May 2000.

[45] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, June 1985.

[46] K. L. Clarkson. Algorithms for polytope covering and approximation. In *WADS '93: Proceedings of the Third Workshop on Algorithms and Data Structures*, pages 246–252, London, UK, 1993. Springer-Verlag.

[47] K. L. Clarkson. Las vegas algorithms for linear and integer programming when the dimension is small. *J. ACM*, 42(2):488–499, 1995.

[48] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, Portland, Oregon, USA, 2004.

[49] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *Proceedings of the 2007 International Conference on Very Large Data Bases*, pages 183–194, Vienna, Austria, Sept. 2007.

[50] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 451–462. VLDB Endowment, 2006.

[51] A. Deshpande and L. Rademacher. Efficient volume sampling for row/column subset selection. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS '10, pages 329–338, Washington, DC, USA, 2010. IEEE Computer Society.

[52] Y. Diao and M. J. Franklin. Query processing for high-volume XML message brokering. In *Proceedings of the 2003 International Conference on Very Large Data Bases*, Berlin, Germany, Sept. 2003.

[53] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *Proceedings of the 2004 International Conference on Very Large Data Bases*, pages 612–623, Toronto, Canada, Sept. 2004.

[54] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[55] M. Drosou, K. Stefanidis, and E. Pitoura. Preference-aware publish/subscribe delivery with diversity. In *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[56] H. Edelsbrunner. *Algorithms in combinatorial geometry*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[57] R. Fagin. Combining fuzzy information from multiple systems (extended abstract). In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '96, pages 216–226, New York, NY, USA, 1996. ACM.

[58] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. ACM.

[59] C. Garrod, A. Manjhi, A. Ailamaki, B. Maggs, T. Mowry, C. Olston, and A. Tomasic. Scalable query result caching for web applications. *Proc. VLDB Endow.*, 1:550–561, August 2008.

[60] M. Grotschel, A. Martin, and W. R. Packing steiner trees: a cutting plane algorithm and computational results. In *Mathematical Programming*, volume 78, pages 265–281, 1997.

[61] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Middleware '04: Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 254–273, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[62] V. Guruswami and A. K. Sinop. Optimal column-based low-rank matrix reconstruction. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 1207–1214. SIAM, 2012.

[63] P. Haghani, S. Michel, and K. Aberer. Evaluating top-k queries over incomplete data streams. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 877–886, New York, NY, USA, 2009. ACM.

[64] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 61–71, New York, NY, USA, 1986. ACM.

[65] J.-S. Heo, J. Cho, and K.-Y. Whang. The hybrid-layer index: A synergic approach to answering top-k queries in arbitrary subspaces. In F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, editors, *ICDE*, pages 445–448. IEEE, 2010.

[66] J.-S. Heo, J. Cho, and K.-Y. Whang. Subspace top-k query processing using the hybrid-layer index with a tight bound. *Data Knowl. Eng.*, 83:1–19, Jan. 2013.

[67] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: a system for the efficient execution of multi-parametric ranked queries. *SIGMOD Rec.*, 30:259–270, May 2001.

[68] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal*, 13(1):49–70, Jan. 2004.

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

[70] M. Jacob and Z. G. Ives. Sharing work in keyword search over databases. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 577–588, New York, NY, USA, 2011. ACM.

[71] M. A. Jaeger, H. Parzyjegla, G. Mühl, and K. Herrmann. Self-organizing broker topologies for publish/subscribe systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 543–550, Seoul, Korea, Mar. 2007.

[72] H. Kaplan, N. Rubin, M. Sharir, and E. Verbin. Efficient colored orthogonal range counting. *SIAM J. Comput.*, 38:982–1011, June 2008.

[73] J. Kleinberg and E. Tardos. Ch 7: Network flow. In *Algorithm Design*. Addison Wesley, 2005.

[74] H.-P. Kriegel, P. Kröger, and A. Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1):1:1–1:58, Mar. 2009.

[75] G. T. Lakshmanan, Y. Li, and R. Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008.

[76] J. Ledlie, P. Pietzuch, and M. Seltzer. Stable and accurate network coordinates. In *Proceedings of the 2002 International Conference on Distributed Computing Systems*, page 74, Vienna, Austria, July 2002.

[77] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 131–142, New York, NY, USA, 2005. ACM.

[78] X. Lu, X. Li, T. Yang, Z. Liao, W. Liu, and H. Wang. Rrps: A ranked real-time publish/subscribe using adaptive qos. In *Proceedings of the International Conference on Computational Science and Its Applications: Part II*, ICCSA '09, pages 835–850, Berlin, Heidelberg, 2009. Springer-Verlag.

[79] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. *Proc. VLDB Endow.*, 1(1):451–462, 2008.

[80] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 2002 USENIX Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts, USA, Dec. 2002.

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

[82] J. Matousek. *Lectures on Discrete Geometry*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[83] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2:169–186, November 1992.

[84] M. Migliavacca and G. Cugola. Adapting publish-subscribe routing to traffic demands. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 91–96, New York, NY, USA, 2007. ACM.

[85] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 635–646, New York, NY, USA, 2006. ACM.

[86] Y. Nekrich. Space efficient dynamic orthogonal range reporting. *Algorithmica*, 49:94–108, October 2007.

[87] T. Neumann, M. Bender, S. Michel, R. Schenkel, P. Triantafillou, and G. Weikum. Distributed top-k aggregation queries at large. *Distrib. Parallel Databases*, 26:3–27, August 2009.

[88] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *Proceedings of the 2002 IEEE International Conference on Computer Communications*, New York, New York, USA, June 2002.

[89] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, New York, NY, USA, 1993. ACM.

[90] D. Papadias and Y. Tao. Reverse nearest neighbor query. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*, pages 2434–2438. Springer, 2009.

[91] O. Papaemmanouil, Y. Ahmad, U. Cetintemel, J. Jannotti, and Y. Yildirim. Extensible optimization in an overlay data dissemination trees. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, USA, June 2006.

[92] O. Papaemmanouil, U. Çetintemel, and J. Jannotti. Supporting generic cost models for wide-area stream processing. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1084–1095, Washington, DC, USA, 2009. IEEE Computer Society.

[93] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic multicast for content-based data dissemination. In *Proceedings of the 2005 International Conference on Data Engineering*, Tokyo, Japan, Apr. 2005.

[94] J. Pereira, F. Fabret, H. A. Jacobsen, F. Llirbat, and D. Shasha. WebFilter: A high-throughput XML-based publish and subscribe system. In *Proceedings of the 2001 International Conference on Very Large Data Bases*, Roma, Italy, Sept. 2001.

[95] D. Powell. Group communication. *Communications of the ACM*, 39(4):50–53, 1996.

[96] K. Pripužić, I. P. Žarko, and K. Aberer. Top-k/w publish/subscribe: finding k most relevant publications in sliding time window w. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, pages 127–138, New York, NY, USA, 2008. ACM.

[97] V. Ramasubramanian, R. Peterson, and E. G. Sirer. Corona: A high performance publish-subscribe system for the World Wide Web. In *Proceedings of the 2006 USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, San Jose, California, USA, May 2006.

[98] P. Rao, J. Cappos, V. Khare, B. Moon, and B. Zhang. Net-χ: unified data-centric internet services. In *NETDB'07: Proceedings of the 3rd USENIX international workshop on Networking meets databases*, pages 1–6, Cambridge, MA, 2007.

[99] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31:161–172, August 2001.

[100] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *NGC '01: Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pages 14–29, London, UK, 2001. Springer-Verlag.

[101] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 2001 IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[102] M. Sadoghi and H.-A. Jacobsen. Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 637–648, New York, NY, USA, 2011. ACM.

[103] M. Sadoghi and H.-A. Jacobsen. Relevance matters: Capitalizing on less (top-k matching in publish/subscribe). *Data Engineering, International Conference on*, 0:786–797, 2012.

[104] S. Shah, K. Ramamritham, and C. V. Ravishankar. Client assignment in content dissemination networks for dynamic data. In *Proceedings of the 2005 International Conference on Very Large Data Bases*, pages 673–684, Trondheim, Norway, Aug. 2005.

[105] D. Shi, J. Yin, Z. Wu, and J. Dong. A peer-to-peer approach to large-scale content-based publish-subscribe. In *WI-IATW '06: Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology*, pages 172–175, Washington, DC, USA, 2006. IEEE Computer Society.

[106] A. Shraer, M. Gurevich, M. Fontoura, and V. Josifovski. Top-k publish-subscribe for social annotation of news. In *Proceedings of the 39th International Conference on Very Large Data Bases*, 2013.

[107] A. Silberstein, K. Munagala, and J. Yang. Energy-efficient monitoring of extreme values in sensor networks. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, Chicago, Illinois, USA, June 2006.

[108] M. A. Soliman, I. F. Ilyas, D. Martinenghi, and M. Tagliasacchi. Ranking with uncertain scoring functions: semantics and sensitivity measures. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 805–816, New York, NY, USA, 2011. ACM.

[109] SportVU. http://www.stats.com/sportvu/basketball.asp.

[110] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[111] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *IEEE Trans. on Knowl. and Data Eng.*, 16:1555–1570, December 2004.

[112] A. Tariq, B. Koldehofe, G. Koch, and K. Rothermel. Providing probabilistic latency bounds for dynamic publish/subscribe systems. In *Proceedings of the 16th ITG/GI Conference on Kommunikation in Verteilten Systemen 2009 (KiVS 2009)*, Kassel, Germany, Januar 2009. Springer.

[113] P. Tsaparas, N. Koudas, Y. Kotidis, T. Palpanas, and D. Srivastava. Ranked join indices. In *In ICDE*, pages 277–288, 2003.

[114] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin Heidelberg, 2003.

[115] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Reverse top-k queries. In *In ICDE*, pages 365–376, 2010.

[116] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Monochromatic and bichromatic reverse top-k queries. *IEEE Trans. Knowl. Data Eng.*, 23(8):1215–1229, 2011.

[117] S. Voulgaris, E. Rivire, A. M. Kermarrec, and M. van Steen. Sub-2-sub: Self-organizing content-based publish and subscribe for dynamic and large scale collaborative networks. In *5th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[118] Yahoo! Search Query Tiny Sample from Yahoo! Webscope. http://webscope.sandbox.yahoo.com/catalog.php?datatype=l.

[119] B. Yang and G. Jeh. Retroactive answering of search queries. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 457–466, New York, NY, USA, 2006. ACM.

[120] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proceedings of the 2003 International Conference on Data Engineering*, pages 189–200, Bangalore, India, Mar. 2003.

[121] A. Yu, P. K. Agarwal, and J. Yang. Generating wide-area content-based publish/subscribe workloads. In *Proceedings of the 5th International Workshop on Networking meets databases (NETDB'09)*, Big Sky, Montana, October 2009.

[122] H. Yu, P. K. Agarwal, R. Poreddy, and K. R. Varadarajan. Practical methods for shape fitting and kinetic data structures using core sets. In *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, pages 263–272, New York, NY, USA, 2004. ACM.

[123] Y. Zhou, B. C. Ooi, and K.-L. Tan. Disseminating streaming data in a dynamic environment: an adaptive and cost-based approach. *The VLDB Journal*, 17(6):1465–1483, 2008.

[124] Y. Zhou, A. Salehi, and K. Aberer. Scalable delivery of stream query result. *Proc. VLDB Endow.*, 2:49–60, August 2009.

[125] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 11–20, New York, NY, USA, 2001. ACM.

Biography

Name: Albert Yu

Date and Place of Birth: May 17, 1984, Hong Kong, China

Education:

Duke University, Durham, NC 08/2006-07/2013. Ph.D. in Computer Science

Polytechnic University, Brooklyn, NY, 05/2006. MS in Computer Science

Polytechnic University, Brooklyn, NY, 05/2006. BS in Computer Science

Selected Publications:

A. Yu, P. K. Agarwal, and J. Yang. ”Subscriber Assignment for Wide-Area Content-Based Publish/Subscribe”, in TKDE’12 (Invited paper to the special issue of the Best Papers in ICDE 2011).

A. Yu, P. K. Agarwal, and J. Yang. “Processing a Large Number of Continuous Preference Top-k Queries”, in SIGMOD ’12.

A. Yu, P. K. Agarwal, and J. Yang. “Processing and Notifying Range Top-k Subscriptions”, in ICDE ’12.

A. Yu, P. K. Agarwal, and J. Yang. “Subscriber Assignment for Wide-Area Content-Based Publish/Subscribe”, in ICDE ’11.

A. Yu, P. K. Agarwal, and J. Yang. “Generating Wide-Area Content-Based Publish/Subscribe Workloads”, in NetDB ’09.

B. Chandramouli, J. Yang, P. K. Agarwal, A. Yu, and Y. Zheng. “ProSem: Scalable Wide-Area Publish/Subscribe”, in SIGMOD ’08.