Continuous top-k queries over real-time web streams

Despoina Vouzoukidou

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Nelly VOUZOUKIDOU

Pour obtenir le grade de
DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Évaluation de Requêtes Top-k Continues à Large-Échelle

Continuous top-k queries over real-time web streams

soutenue le 17 septembre 2015

devant le jury composé de :

M.	Bernd Amann	Directeur de thèse
M.	Vassilis Christophides	Co-encadrant
Mme	Sihem Amer-Yahia	Rapporteur
Mme	Evaggelia Pitoura	Rapporteur
M.	Ludovic Denoyer	Examinateur
M.	Themis Palpanas	Examinateur
M.	Dan Vodislav	Examinateur

Abstract

In the era of the real-time web, online news media aggregators, like Google News, and social media sites, like Twitter, strive for effective and efficient filtering of large volumes of information for millions of users. Given the vast diversity and burstiness of information published on the web each minute, filtering, ranking and delivering content streams to interested users becomes a challenging task. Moreover, feedback signals on the content, such as clicks or shares, provide useful information on content importance, but also require more complex manipulation techniques for filtering these streams. Scoring functions in this context consider both static and dynamic ranking factors, such as profile relevance, recency of information and dynamic feedback signals. Existing works for the real-time web fail to handle such dynamic scoring functions in an online way and rather adapt an approach of iterative execution of snapshot queries.

In this thesis, we are interested in efficient evaluation techniques of continuous top-k queries over text and feedback streams featuring generalized scoring functions which capture dynamic ranking aspects. As a first contribution, we generalize state of the art continuous top-k query models, by introducing a general family of non-homogeneous scoring functions combining query-independent item importance with query-dependent content relevance and continuous score decay reflecting information freshness. Our second contribution consists in the definition and implementation of efficient in-memory data structures for indexing and evaluating this new family of continuous top-k queries. Our experiments show that our solution is scalable and outperforms other existing state of the art solutions, when restricted to homogeneous functions. Going a step further, in the second part of this thesis we consider the problem of incorporating dynamic feedback signals to the original scoring function and propose a new general real-time query evaluation framework with a family of new algorithms for efficiently processing continuous top-k queries with dynamic feedback scores in a real-time web context. Finally, putting together the outcomes of these works, we present MeowsReader, a real-time news ranking and filtering prototype which illustrates how a general class of continuous top-k queries offers a suitable abstraction for modelling and implementing continuous online information filtering applications combining keyword search and real-time web activity.

Contents

List of Figures

1 Introduction

1.1 Context: Web 2.0

The emergence of *Web 2.0* technologies over the past decade has transformed the Web into a vibrant information place of content exchange, collaboration and communication. The first generation of the World Wide Web, often referred to as *Web 1.0*, was designed as a "read-only" environment of hyperlink-connected web pages and documents. While the Web could be seen as a huge public library, the role of an end-user was limited to *accessing* information through a browser by directly submitting a URL, following links or by using a search engine. Web 2.0 has drastically changed this perception of the Web by assigning an active role to end-users: from passive content consumers, they can nowadays actively participate by generating content themselves and by expressing their opinion and giving *feedback* on published information.

Social networks (e.g. Facebook[1]), blogs and microblogs (e.g. Twitter[2]), photo, music and video sharing platforms (e.g. Flickr[3], YouTube[4]) and review and rating websites (e.g. Yelp[5]) are only a few examples of *social media* applications favoring user generated content (Figure 1.1). Such applications become extremely popular since they enable a human-centric sensing of a large spectrum of our everyday life activities in professional, residential and public spaces, covering for example users needs for information, communication, entertainment and shopping. Statistics show that in 2014, social networking and micro-blogging represent more than 40% of an average user's online activities[6].

At the same time, traditional sources of information, such as newspapers, television and radio, have a strong presence on the Web. In fact, news reading remains one of the most popular activities online, with users preferring online news media websites to print press[6]. While vast amounts of user generated content becomes available

[1] www.facebook.com
[2] www.twitter.com
[3] www.flickr.com
[4] www.youtube.com
[5] www.yelp.com
[6] GlobalWebIndex: http://insight.globalwebindex.net/hs-fs/hub/304927/file-1414878665-pdf/Reports/GWI_Media_Consumption_Summary_Q3_2014.pdf

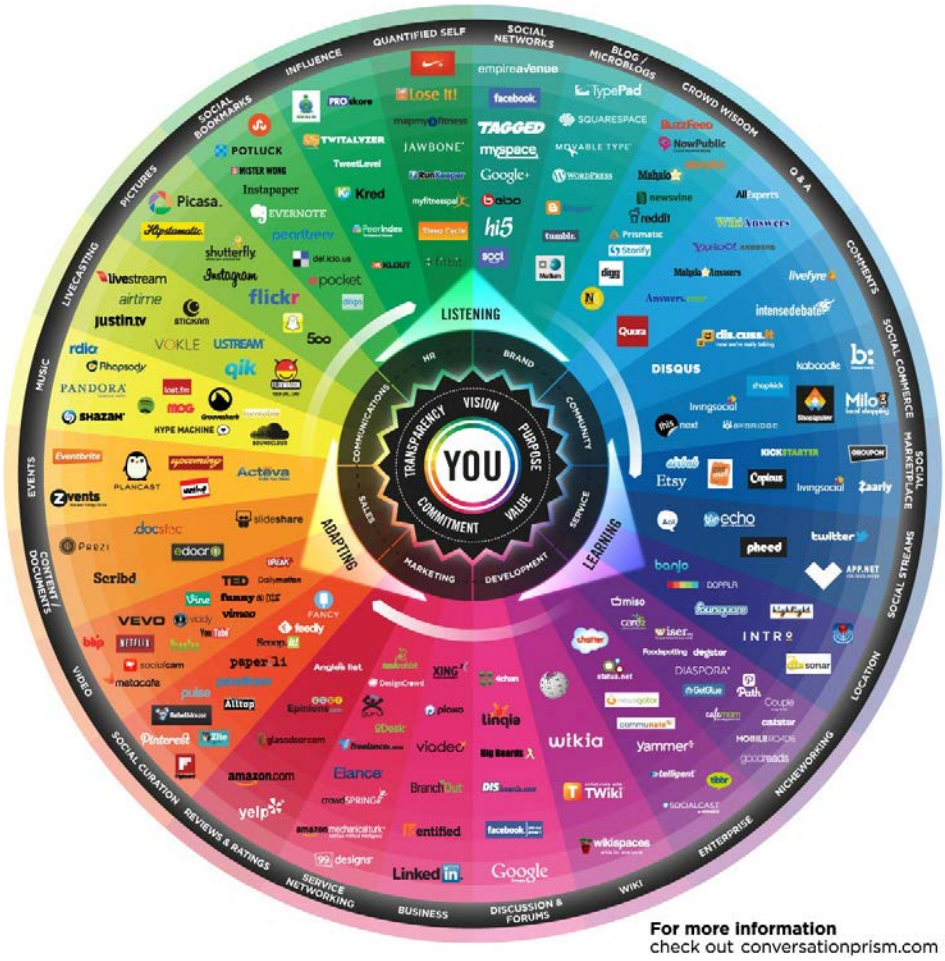


Figure 1.1: The Conversation Prism (conversationprism.com), depicting the most popular social media, categorized by their usage

every minute in social media, traditional information producers, provide users with all necessary tools and encourage them to lively participate by commenting, sharing and generally expressing their opinion over published information. This interaction is mostly supported by social media applications, strongly or loosely coupled with news media sites and provides valuable feedback to assess the importance of published information. Additionally, adapting to the *real-time web* era, which requires information to be made available immediately as it happens, online news media continuously publish novel information and updates on real life events throughout the day.

1.2 Accessing information in Web 2.0

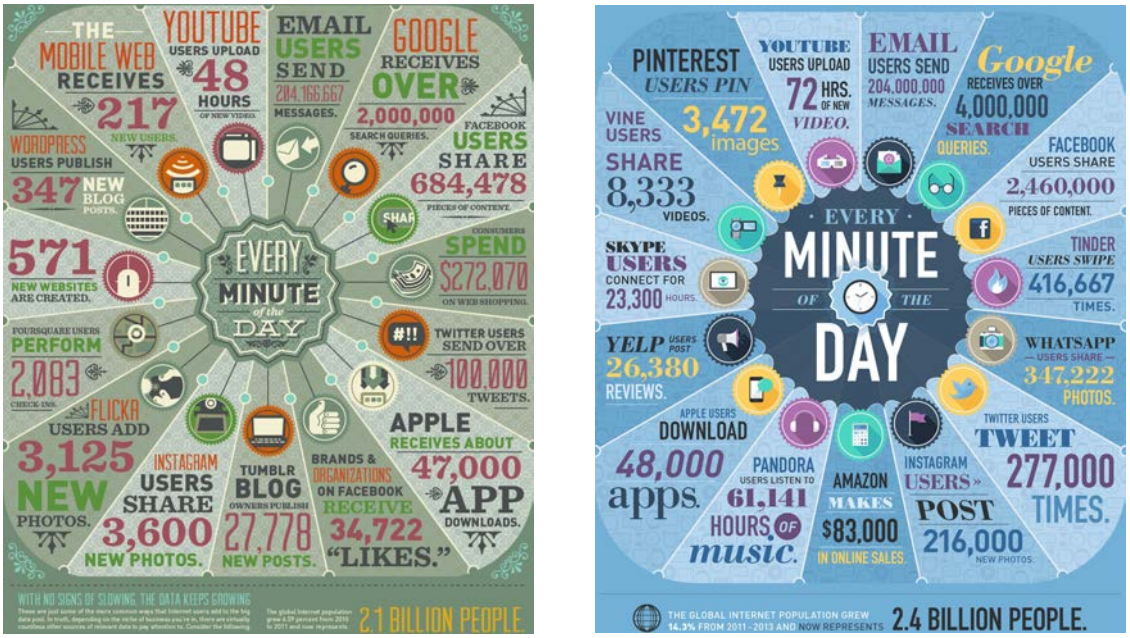
With large amounts of information and feedback on information being published each minute and with millions of news and social media users expecting to receive novel information in real-time, the initial role of the Web, that of *accessing* information, poses new challenges. Figure 1.2 depicts the information overflow effect: more information is published every minute that we can actually consume. While searching over Web 1.0 required answering queries over a slowly evolving database of static web pages, nowadays the streaming nature of Web 2.0 contents creates a vital need for achieving an effective, *near real-time information awareness* for millions of users. In this context of searching over dynamic streams of Web 2.0 information, research issues that have been raised include *continuous information filtering* [MP11, HMA10, VAC12], *trends detection* [MK10, CDCS10, HMA12], *content recommendation* [MZL+11, ZXL+12] and *indexing of streaming data* [CLOW11, WLXX13].

1.2.1 News aggregation and the RSS syndication

In the context of news media, back in Web 1.0, when a limited number of sources was available online, it was feasible for simple users to visit directly the news websites of their preference and read any news updates. Today, given the increasing number of news sources and the high rates at which each of these publish information, this approach is almost impossible.

Online news *aggregation systems* like Google News[7], Yahoo! News[8] or MSNBC News[9], but also blog search engines such as Google Blog Search[10] undertake the role of collecting these streams of information from a large number of sources, ranking them and making them available for millions of users. More importantly, they can provide *personalized* views over the streams, using information explicitly given by the users, such as their interests (keyword queries) or implicitly collected by monitoring their behavior (clicks, feedback). Personalization might go beyond user preferences and take account of user background (e.g. friends in social networks) as well as contextual

[7] news.google.com
[8] news.yahoo.com
[9] www.msnbc.com
[10] www.google.com/blogsearch



(a) (b)

Figure 1.2: User generated content each minute in (a) 2011 and (b) 2013 (*source: https://www.domo.com/learn/data-never-sleeps-2*)

information in which a user interacts with the system (e.g. location, time and device characteristics).

The *RSS syndication* (*Really Simple Syndication* or *Rich Site Summary*), which became popular in the early 2000s, allowed the timely diffusion of streaming information to users. The idea behind RSS is that the user should be *notified* about new information as soon as it becomes available, instead of having to visit a website (e.g. an original news source or an aggregator) and manually check for updates. An *RSS feed* is a document containing several *items*, i.e. information chunks, each of which has a title, a summary and some metadata, including the publication date and a link to the source. New items are inserted in the RSS feed by the publisher, while older ones can be removed. Users can *subscribe* to such feeds using an *RSS reader* (e.g. feedly[11]), which undertakes the task of checking and delivering updates. RSS and Atom, a similar syndication for the delivery of streaming data, are supported by the majority of news media and news

[11] www.feedly.com

aggregators today.

While the RSS and Atom syndication provide the means for immediately notifying users on fresh information, its use alone cannot provide personalized views on information. The *Publish/Subscribe* messaging paradigm models the *continuous approach* of filtering RSS, social media or other, similar data streams [EFGK03]. In the *topic-based* variation, publishers categorize contents into predefined, static topics and insert each item (e.g. news article item) in the corresponding feeds of its topics. For example, in an American news media website, an article on Barack Obama making a statement on Europe, would probably be inserted in the feeds on "*politics*" and "*international news*". A user interested in either of these topics, can subscribe to the corresponding feeds and receive related information. In the *content-based* variation of Publish/Subscribe, instead of predefining topics of interest, users can subscribe by issuing queries of their own choice. These queries, also called *subscriptions*, serve as filters for incoming information. Chapter 2 gives a thorough overview of how subscriptions have be defined in the literature and the models that have been proposed on how to filter them against streaming information.

1.2.2 Real-time web

In the era of the *real-time web*, social media platforms such as Twitter, provide users the means to instantly publish information on it emerges. The nature of messages on Twitter (tweets), having a small length of less than 140 characters, and the ease of publishing them from any portable device, facilitate the real-time publication of events. It is common today that information on real life events, such as an earthquake, is published on Twitter by simple end-users, before it has been published in news media [SOM10]. The volume of information published on Twitter has drastically increased over the past few years: from a publication rate of merely 200 tweets per minute in Twitter's early years (2008), Twitter reached 24,000 posts per minutes in 2010 and 236,000 in 2012. Today, this number has exceeded the 340,000 tweets per minute.

While information is nowadays published on real-time, there is a vital need for providing efficient and effective filtering over these streams. Additionally, feedback on items, such as *shares* (retweets) and *replies*, provide useful information on content

importance, but also require more complex manipulation techniques for filtering the streams. Scoring functions in this context, used for the ranking items for millions of users, consider both static and dynamic ranking factors, like profile relevance, recency of information and dynamic feedback signals. So far, works over real-time search, both in the literature and in commercial systems, fail to handle such dynamic scoring functions in an online way and rather adapt an approach of *periodic execution of snapshot queries*. In Chapter 4 we will show how this kind of dynamic ranking scores can be handled online with time and memory efficient techniques.

1.3 Approach: Continuous top-k queries over text streams

In this thesis we are particularly interested in the continuous, scalable filtering of dynamic text streams for a large number of users. Similarly to the content-based filtering in the Publish/Subscribe model, we consider that a user can issue a number of keyword queries in order to retrieve information items of interest from different streams. The system undertakes the task of storing these *continuous queries* and evaluates them against arriving items. To perform this evaluation, we follow the *continuous top-k textual query evaluation* approach [MP11]: an underlying scoring function determines the relevance for each query-item pair. Our goal is to maintain for each query the list of the k most highly ranked items at every time instant. Additionally to incoming items, we consider that feedback signals on items (*events*) also become available in the stream.

To effectively filter the stream and provide personalized views over it, the underlying scoring function should consider for each item, both *query-dependent content relevance* and *query-independent item importance*. Standard relevance scores are cosine similarity and Okapi BM25 [RWJ+95]. Item importance can be estimated by static measures, like information novelty [GDH04], source authority [DCGR05, HLLM06, MC10, MLY+10a], content diversity [DSP09] or dynamic feedback signals, like user attention [WZRM08, LDP10]. Moreover, to take into account information freshness, newer items are usually considered as more important than older ones. This functionality is supported either by using time decay functions [CSSX09] or by sliding window techniques.

For instance consider the example of news media streams. Suppose that a user is interested in being informed about natural disasters and submits a continuous query on "*earthquakes*". When the earthquake struck Nepal in April 2015, thousands of news media reported on the event and continuously published updates on its consequences. These huge amounts of news articles could overwhelm the user, even though the information within was relevant to the submitted query. Following the continuous top-k query evaluation approach, the user would only receive the items (news articles) that at any time instant, were ranked in the top-k list of the query. The underlying scoring function should consider the query-dependent textual relevance between the query and the item, but also additional parameters like the source authority. For instance, an article published in the New York Times should be considered as more important than another one published in a newly created website with a small number of visitors. While on the arrival of an item, we can have a first estimation of its importance, feedback signals from other users, e.g. clicks or shares, should additionally be exploited to re-evaluate its query-independent item importance. Aiming at providing real-time information, events on items should also be handled on their arrival.

Figure 1.3 depicts an abstract overview of our proposed functionality for the real-time filtering of dynamic information streams. Each incoming *query*, *item* or *feedback event* can trigger the *update of a query*, i.e. alter its top-k list of items. When a query is initially issued, a *query handler* component evaluates it and returns to the user the list of the k most highly ranked items. This query is also stored in an *item handler* component. As new items are published in the stream, they are evaluated against stored queries in the item handler component which retrieves potential query updates and inserts the item to the corresponding queries' top-k lists. An additional *event handler* component undertakes the task of checking for potential updates when the dynamic score of an item changes.

1.4 Contributions

In this thesis we aim at providing efficient and scalable solutions for the evaluation of continuous text queries over item and feedback streams. In a nutshell our main contributions are the following:

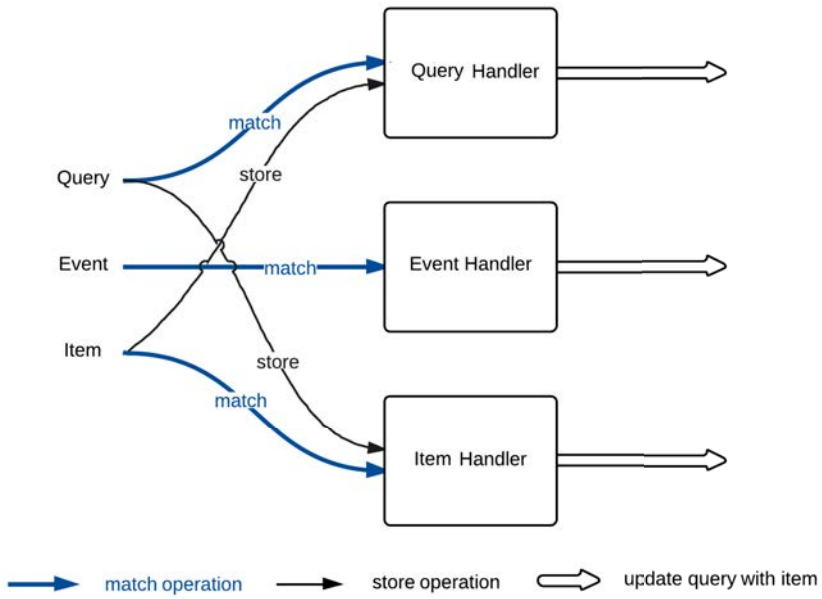


Figure 1.3: Our proposed continuous query evaluation model

- As a first contribution, we have proposed a generic framework for defining different families of continuous top-k queries, handling highly dynamic streams of items and feedback signals. The core of this framework is a generic scoring function which captures both static and dynamic ranking criteria like content relevance, information freshness, source authority, information novelty, user feedback etc. for filtering information over text streams.

- Based on this framework, we propose a new continuous top-k query evaluation approach going beyond the state of the art solutions. Existing solutions [MP11, HMA10] only consider content relevance (e.g. cosine similarity or Okapi BM25) and do not support query-independent scoring parameters, like source authority etc. On the other hand, existing real-time web filtering solutions [CLOW11, BGL+12, WLXX13, MME+14, LBLT15], do consider dynamic scoring functions as described previously, but fail to handle dynamic item and event streams in an online way. They focus instead on the efficient indexing of new items (the query handler component) and periodically execute snapshot queries in order to handle long-running queries. Our solutions are based on novel algorithms, which allow the definitions of search bounds which drastically prune the search space

for processing items and feedback events. We also provide formal proofs for the optimality and soundness of these bounds.

- We have implemented our continuous top-k query evaluation approach. In our implementations, we adapt a number of in-memory data structures such as dictionaries (i.e. inverted indexes), ordered lists and spatial indexes. We provide a thorough experimental evaluation of these implementations using real data from RSS feeds and from Twitter and compare their memory/matching time trade-offs. We show the scalability of our approach over the number of stored queries and the arrival rates of items and feedback signals. We also show that our methods are more efficient than existing state of the art solutions on continuous queries.

- Finally, we present MeowsReader, an online news recommendation prototype putting together the results of our work. The current running version of MeowsReader collects news article items from more than a thousand general and specialized U.S. news media RSS feeds and converts user clicks and tweets from Twitter's public APIs into item feedback scores.

1.5 Organization of this thesis

The rest of this thesis is organized as follows:

Chapter 2 conducts a survey on the related work. We first present the filtering techniques and ranking models that have been proposed so far in the literature and applied in commercial systems for query filtering over information streams. Focusing on the top-k ranking approach, we present the most important ranking parameters and scoring functions employed in such contexts. We then, describe the state of the art algorithms and data structures which are directly comparable to our work.

Chapter 3 describes our first approach on evaluating continuous top-k queries [VAC12]. Considering a generalised scoring function and extending state of the art solutions, we present a novel method on tackling the continuous query evaluation problem. We experimentally evaluate the performance of our proposed indexes and additionally propose a probabilistic model over our solution, which can be exploited to provide an approximate solution.

In Chapter 4 we introduce our real-time search algorithms, implementing the Event Handler component of our proposed model (Figure 1.3). To the best of our knowledge, this is the first attempt of processing dynamic scores in a continuous setting. Our experiments show that our proposed solution can handle the high arrival rates of a real-time web system, such as Twitter.

Chapter 5 presents MeowsReader our news recommendation prototype putting together the results of this thesis [VAC14]. This prototype is available online (at gateway.lip6.fr:8080/meows/) and integrates our solutions with some additional features like a simple trend detection mechanism.

Finally, in Chapter 6 we conclude our work and present a list of open scientific questions and future work related to this thesis.

2

Related Work

The main goal of this thesis is to propose in-memory data structures and algorithms for the efficient, online evaluation of continuous (long running) queries over text streams. Formulated in different ways, this problem has been studied and applied in a wide spectrum of contexts ranging from relational databases to online news aggregators and more recently, in real-time micro-blogging systems. The terms *continuous queries*, *subscriptions* in the Publish/Subscribe model, *alerts* in commercial systems such as Google News alerts[1], and also *real-time search*, all refer to the same notion: the filtering and dissemination of information from data streams to interested users, as soon as it becomes available. In this chapter we present the state of the art solutions proposed in the literature and applied in commercial systems. We make the distinction between two main approaches: the *continuous approach*, in which user queries are indexed and evaluated with information from the stream immediately on its arrival and the *periodic execution of snapshot queries* approach, in which incoming information items are indexed and periodically queried in the database.

A crucial factor in the formalization of the continuous query evaluation problem and the solutions proposed, is the underlying *filtering technique*, i.e. the conditions under which an item from the stream is inserted into a query's result set. After presenting the most important filtering techniques, we focus on the *continuous top-k ranking model*. In this model, the definition of a scoring function assigning a score to each query-item pair is also required. The definition of an adequate scoring function depends both on the quality of information streams, but also on the expected value of obtained results in concrete applications contexts. In order to propose a generic solution to the continuous top-k query evaluation problem, we study several ranking parameters proposed to accommodate specific application needs and abstract from these a generalized function capturing both static and dynamic aspects of information arriving in streams.

This chapter is organized as follows: Section 2.1 presents the most important filtering techniques employed over streams of information, ranging from the boolean model to the continuous top-k ranking model. We also discuss the two main approaches in

[1] www.google.com/alerts

capturing the temporal aspect of streaming information, namely data windows and decay functions. In Section 2.2 we conduct a survey on the most important scoring functions and ranking parameters proposed so far in the literature. Finally, in Section 2.3 we present the state of the art indexing techniques comparable to our work.

2.1 Filtering techniques and ranking models

Ever since its first definition, the continuous query evaluation problem has been applied in several contexts. This section gives an overview of the most important filtering techniques used over structured data, text and social media streams. Such techniques include the *boolean model*, *ranking models* and *recommender systems*.

2.1.1 Filtering over structured data

The notion of *continuous queries* was first introduced in 1992 in *Tapestry* [TGNO92], an email database system, to denote long-running, SQL-like queries over an append-only relational database. The result of a continuous query was defined as "*the set* [union] *of data that would be returned if the query was executed at every instant in time*". Observe that given this definition, once a tuple is inserted in a result set it will not be removed at any future time instant, even if it no longer fulfills the query's requirements.

In a simplified context, we can imagine that a tuple (item) can be tested against stored continuous queries and potentially be inserted in their result set, only on its arrival. In the following, we use the term *simple filtering* to refer to such methodologies where each item is handled only on its arrival and results do not depend on previously arriving content.

However, the general form of queries supported by Tapestry (Figure 2.1), using time constraints, joins and the existential operator, goes beyond simple filtering: a tuple can potentially fulfill a query's requirements some time *after* its arrival. For instance, consider a stream of emails and the continuous query "*select emails with at least one reply*": no email will be inserted in the query's result set on its arrival, as it obviously does not have any replies at that time. However, the arrival of a reply will trigger the insertion of the original email in the results.

select ...
from $tbl1$, $tbl2$, ...
where (C_{11} **and** C_{12} **and** ...) **or**
(C_{21} **and** C_{22} **and** ...) **or** ...

Figure 2.1: General form of continuous queries supported by Tapestry. C_{ij} represent conditions such as comparisons (e.g. $tb1.att > 4$) or existential operators ($not\ exists(...)$)

Ranking models

The work in [TGNO92] marked the beginning of a new model of data processing: the *Data Streams Management Systems* (DSMS), which generalized Database Management Systems (DBMS) for the processing of continuous queries over structured data streams [BBD+02]. However, it was not until 2005 that quality of results was also considered [LYWL05]. Items that matched a continuous query were ranked and only the "best matching" items from the stream would be included in the query's result set.

The first ranking model proposed for the continuous queries context was given in [LYWL05] and [TP06] where *continuous skyline queries* were introduced (Figure 2.2b). Defining a query as a set of dimensions to optimize (e.g. maximize or minimize), the skyline of a query is the set of tuples that are not *dominated* by any others w.r.t. that query. We say that a tuple t dominates another tuple t', if and only if t is preferable to t' in every dimension of the query. In other words, a tuple t belongs in the skyline of a query if there exists no other tuple that is better than t in *all* the query's dimensions. The result of a *continuous* skyline query is the set of tuples that form its skyline at any given time instant.

Going a step further, [MBP06] introduced the notion of *continuous top-k queries* (Figure 2.2c). The ranking by skylines was replaced by a linear scoring function determining the relevance between a query and an item. The continuous top-k query evaluation problem was defined as the maintenance of the list with the k most highly ranked items (top-k) w.r.t. each query at every time instant.

Comparison

In the previous we have presented four categories of filtering techniques over streaming structured data, namely the *simple filtering*, the *SQL filtering* of Tapestry and the *skyline* and *top-k ranking models*. As we will see later, these and similar techniques are also applied over text and social media streams.

Among these methodologies the most straightforward is the simple filtering technique: items are only considered on their arrival and since they do not affect the future results of any query, they do not need to be indexed. However, given the restriction of not relying on past items, the types of supported definitions of continuous queries are limited. For example, in databases, joins cannot be supported.

The SQL approach proposed in Tapestry can support more complex queries by considering the whole history of past items. Another difference of this approach w.r.t. simple filtering and to ranking models, is that items can be inserted in result sets at time instants future to the item's arrival. Such insertions can be triggered, for example, by the arrival of another item.

A main drawback of both simple filtering and the SQL filtering models is that in case of high rates of arrival in the stream or bursts of information, users can be overwhelmed with the number of results received, as all items matching their queries are inserted in the result sets. The notion of ranking introduced in the skyline and top-k ranking models, solves this problem by applying a further filtering of results: additionally to fulfilling a query's requirements, items need to be considered among the "best matching" ones in order to be inserted in a query's result set. Another difference from previous techniques is that the goal is no longer the simple insertion of items in a result set of a query, but rather its maintenance, as deletions might also occur.

A first difference between the continuous skyline and the continuous top-k ranking models is on the number of results. While in the top-k approach the size of the result lists is bounded by the k parameter, in the skyline approach, this size can vary depending on the values of items in each of the query's dimensions. In terms of quality or results, the use of skyline queries implies that all dimensions of the query are equally important. In top-k queries on the other hand, the underlying scoring function determines the importance of each of the dimensions. For instance, in the case of linear scoring functions, the coefficients of the function determine the importance of each

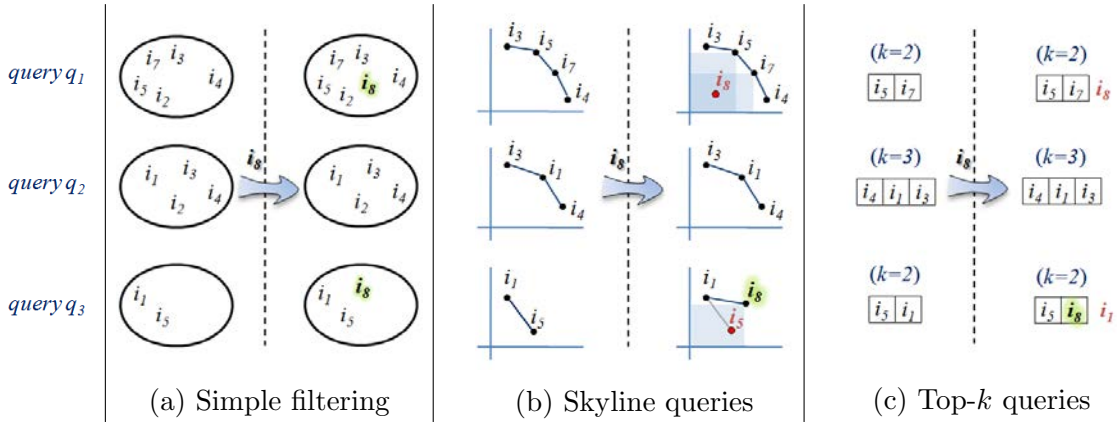


Figure 2.2: Filtering techniques: The result set of three queries over the (a) simple filtering model and the (b) skyline and (c) top-k ranking models

variable (dimension). Finally, top-k queries provide a total order of results in the top-k lists, while in skyline queries all items are considered to be in the same rank.

To illustrate the differences between these filtering methodologies, Figure 2.2 shows an example of three queries q_1, q_2 and q_3 and their results sets, right before and after the arrival of an item i_8 over the simple filtering model, the skyline (supposing there are only two dimensions in the query) and top-k problem formulations. Item i_8 is only relevant to queries q_1 and q_3, so q_2 is never affected by this item's arrival. In the case of simple filtering, both of these queries receive i_8 in their result set. However, this is not the case when a ranking model is applied. In continuous skyline queries, we can see that i_8 is dominated by both i_5 and i_7 which were already in the result set of q_1, so i_8 will not be inserted. On the other hand, i_8 belongs in the skyline of query q_3, so it is inserted in the corresponding result set, and also triggers the deletion of i_5 which is now dominated by i_8. Finally, in the continuous top-k query evaluation scenario, only the k most relevant items are stored for every query. For query q_1, the new item i_8 is ranked lower than item i_7 and thus, is not inserted in the top-k list. For query q_3, however, i_8 is ranked second, so it is inserted in the result set, and also triggers the deletion of the previously ranked as k-th (second) element, i_1. Observe that k can have different values for different queries.

2.1.2 Filtering over text streams

Soon after the first definition of continuous queries over relational databases, in [YGM94] the authors proposed an adaptation of continuous queries over unstructured data: streams of tuples are replaced by documents (*items*) streams and SQL (or SQL-like) queries, by user defined *text queries*, also referred to as *subscriptions*. Initially, a *boolean model* was employed: a query is defined using conjunction, disjunction and negation operators over terms, and its result is the set of all matching items, without any ranking among these.

This model was later extended in *SIFT* [YGM99] (further analyzed in Subsection 2.3.2) to consider text similarity between query-item pairs. For each query, a static threshold score is assigned. An incoming item containing some of the terms of the query (disjunction semantics) is inserted in the query's result set if and only if the similarity score exceeds the query's threshold. Queries and items are represented using the Vector Space Model (VSM) and cosine similarity is used to evaluate query-item similarity. Observe, that despite the use of a scoring function between queries and items, this is still a simple filtering problem: all items that match a given query and exceed its threshold belong in its result set and the ranking implied by the scoring function among items does not affect the results.

The development of Web 2.0 technologies and the need for filtering huge amounts of information has encouraged the adaptation of existing filtering techniques over text streams. The more recent work in RoSeS [CATV11] presents a general framework for the filtering of continuous queries over semi-structured data. Aggregating a number of different streams of information (RSS feeds), RoSeS supports continuous queries with preferences on the sources (e.g. only accept *BBC* and *New York Times* as sources), equality operators on given fields of RSS items (e.g. *date="17-09-2015"*) and also joins over items of different sources (e.g. pairs of items from *BBC* and *New York Times* that have the same/a similar title). Going a step further, continuous queries can also support conditions on textual information on the title or the description of an item using the boolean model over terms (e.g. items that contain terms A and B).

Ranking models

Also in the context of Web 2.0, [PZA08] proposed the use of the continuous top-k ranking model over text streams. Similarly to the definitions over structured data, a scoring function determines the relevance between any query-item pair and only the top-k items of each query form its result set. This work, as well as several others that followed [MP11, HMA10] (discussed in Section 2.3) considered cosine similarity to define the relevance between a query and an item. Our work in [VAC12] (see Chapter 3) also filters text streams using the continuous top-k approach and was the first one to consider more generalized scoring functions supporting a wide range of ranking parameters proposed in the literature.

The increasing popularity of the so-called real-time web [DFMGL12] over the past few years and the massive amounts of information produced over such streams created new challenges in filtering information for interested users. In an effort towards improving quality of results, Twitter[2], the most important commercial system providing *real-time search* also employs a top-k ranking approach, but uses more complex scoring functions additionally considering dynamic feedback signals available on the items, such as the number of views, 'favorites' or shares (retweets) the item receives [BGL+12]. This approach is further discussed in Section 2.3.

Recommender systems

Recommender systems [GNOT92, Kar01] represent a different way of filtering information, with queries being replaced by *user profiles* built by the system. User profiles, which are based on each user's past behavior (e.g. clicks) can be used independently to filter content for this user (*content-based filtering*) or can be compared to the profiles (behavior) of other users in the system in order to recommend to similar users, similar items (*collaborative filtering*). Following this approach, [LDP10] proposed a collaborative filtering mechanism based on users click behavior to filter news content over the news aggregator, Google News[3].

Recommender systems are generally based on machine learning techniques and the

[2] www.twitter.com
[3] news.google.com

ranking of items is not exposed to the user. Another main difference from the continuous top-k ranking model in the definition of a query: instead of a user explicitly defining, e.g. a text query, the system builds the user profile, based on the behavior. Also observe that, unlike queries, user profiles can dynamically evolve over time, as user behavior changes.

2.1.3 Filtering over social media streams

Over the past decade, social media such as Twitter, Facebook[4], Google+[5] or LinkedIn[6] have become increasingly popular. In the case of social networks, users create explicit connections between them. In Facebook, for instance, users can add each other as "*friends*" indicating, that generally[7], when content is published by each of these users, the other one can read and interact with it (e.g. 'like' or comment). Considering users as nodes and connections as edges, this is often called a social graph. While on Facebook the social graph is undirected, Twitter employs a directed graph approach, with "*followers*" only receiving content from their "*followees*".

In these systems, independently of its definition, the underlying social graph of user connections can provide useful information for effectively filtering published results. A straightforward approach in such systems would be to send to users all content published by their connections. However, with the high popularity of social networks and the huge amount of data published every day, users would be overwhelmed if no further filtering was applied.

Ranking models

In Earlybird, the search tool of Twitter, a top-k ranking model is employed [BGL+12]: users can send their queries to the system, which returns and periodically re-evaluates for them the most highly ranked results. Additionally to a list of other parameters, the social graph of a user (followers and followees) is also considered in the scoring function

[4] www.facebook.com
[5] plus.google.com
[6] www.linkedin.com
[7] Users can also express more complex privacy settings for a published item, like making it visible by a subset of friends or to the whole web

when results are computed. Facebook, considering user profiles to filter information, also considers a top-k modeling scheme for computing results sent to users. This function, called EdgeRank[8] considers among others, the relevance between two users and feedback signals on items.

The more recent work in [AVB15b] presents a general framework for handling continuous top-k queries over social network streams. The employed scoring function considers content similarity between each query-item pair, a user-dependent score computed over the social graph of the user, query and user independent item importance and finally, temporal aspects of information with the use of a decay function. Adopting the Publish/Subscribe model they aim at handling incoming items and events on items as they arrive. On the other hand, other types of incoming information or modifications, such as changes of connections in the social graph, are assumed to have a smaller impact on the top-k query results and are only periodically processed. Some preliminary results of this approach are presented in [AVB15a].

Recommender systems

Recommender systems [GNOT92, Kar01] represent, maybe the most popular approach when filtering content over social networks: users expect to receive information based on their profile and on content created and reviewed by their network. *Collaborative filtering*, which relies on system-inferred user profiles, while additionally considering similarity among users is commonly used in this case [KSJ09]. However, in the following, we will not discuss in detail any collaborative filtering algorithms, as this type of filtering goes beyond the scope of this thesis.

2.1.4 Continuous queries vs. periodic snapshot query execution

There are two main strategies in answering long-running queries proposed in the literature and applied in commercial systems namely, the continuous approach, aiming at maintaining the correct results of the query at any time instant, and the periodic execution of the equivalent snapshot queries.

[8] sproutsocial.com/insights/facebook-news-feed-algorithm-guide/

In the *continuous approach*, which is also the focus of this thesis, a *push model* is employed: as incoming information arrives, it is immediately handled and corresponding queries result sets are updated as necessary. In this event-driven model, query results are always up-to-date and no results are missed. Additionally, it permits the immediate (real-time) notification of users on fresh information items that have been inserted in their queries result sets.

In the approach of *periodic execution of snapshot queries*, which is based on the *pull model*, incoming information (items and feedback) is indexed and results of snapshot queries are only retrieved on request. In this case, the system usually undertakes the task of periodically executing users' stored (long-running) queries and sending them the new results from the top-k lists.

Twitter's search and Alert systems (e.g. Goolge Alerts[9]) are examples of periodic execution of snapshot queries. In the case of Twitter, once a user inputs a query, a first list of results is immediately given as a response. This list is later periodically updated with newer items (tweets). Based on observations on Twitter's website, the top-20 results of user queries are updated every 30 seconds. Google Alerts, can be used over news streams (Google News[10]) or over research publications (Google Scholar[11]). In both cases, a user subscribes giving a text query and is then periodically informed (e.g. once daily) about fresh relevant items, i.e. news articles or scientific publications.

Even though strategies have been proposed in the literature on determining how often a system should perform the updates (re-execute the snapshot queries) [HAA12], there still cannot be any guarantee that when using the periodic execution approach, no results will be missed (Figure 2.3). On the other hand, using a naïve strategy of re-evaluating *all* stored queries' results on every item arrival would be extremely inefficient. On the contrary, the continuous approach on query evaluation guarantees that no results will be missed and that all updates are retrieved immediately, on the item's arrival, and focuses on efficiently filtering the number of stored queries that should be tested for potential updates by each arriving item.

[9] www.google.com/alerts
[10] news.google.com
[11] scholar.google.com

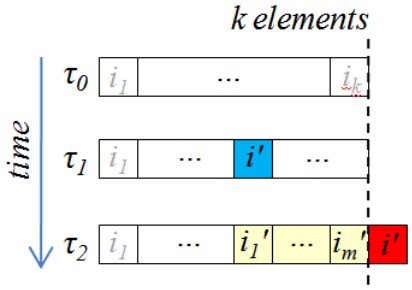


Figure 2.3: The top-k results of a query at three different time instants, τ_0, τ_1, τ_2. At τ_0 item i' is inserted in the result set and is then removed at τ_2, after the insertion of a number of items i'_j. Using the periodic snapshot query execution strategy, if the result was evaluated only at τ_0 and τ_2, item i' would never be retrieved.

2.1.5 Modelling recency of information

One of the most important quality dimensions of information arriving in stream is *freshness*. In most application settings, old data is considered less important or relevant and could skew results from new trends or conditions. As a matter of fact, delivering the most highly ranked items from a stream to any query (or user), without considering how recent these items are, the result set would at some point converge and no new items would be inserted. Two main approaches have been proposed in the literature for addressing old data in data streams, namely, *data windows*, in which items are considered for processing as they appear and *aging models*, in which items are associated with "weights" that decrease over time.

Data windows

A data window represents a finite interval of a data stream. As we can see in Figure 2.4a only the most recent items from the stream are included in a data window for processing. The result sets of all continuous queries stored in the system should only contain items from the *window*, i.e. this set of recent items. In the *sliding windows* variation, older items are continuously removed from the window, while new ones are inserted. In the *tumbling windows* variation, the defined windows are disjoint. In either case, windows can be *time-base* or *count-based*. In the first case, items published in the past w time units are considered for the results, with w being some system pre-defined constant.

In other words, items are assigned on their arrival, an expiration date after which they should be removed from the result set of any query that contains it (if any). In count-based windows, the idea is similar, but instead of considering time, the w last items are always maintained in the window.

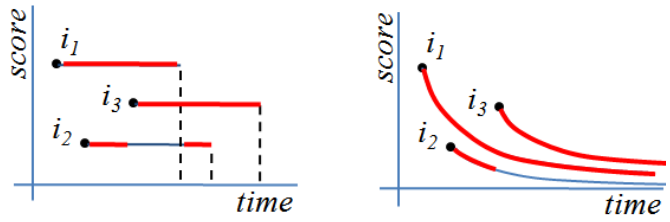
Window semantics have been widely used on streaming structured data [LYWL05, TP06, MBP06, PZA08]. There are also a few works proposing algorithms and data structures for the processing of continuous queries over text streams [MP11, HMA10]. A major advantage of this approach is that all scores assigned remain constant until the item's expiration, which simplifies their processing.

Decay functions

Decay functions consider recency of information by continuously decreasing all scores after their initial assignment (Figure 2.4b). Unlike in the windows approach, items do not expire, but rather "fade out" over time: as their score w.r.t. queries decreases, it becomes easier for newer items to be ranked higher and less probable for older items to appear in any result sets.

A large majority of works studying scoring functions over text streams, like news streams [DCGR05, WZRM08] or real-time micro-blog streams [CLOW11], consider decay functions when processing dynamic streams. This is due to the explicit handling of the time "weight" of items compared to an implicit handling in data windows. To better understand this, consider the example over news streams: suppose that an important real-life event occurs and many news articles are written on the subject over a small period of time. Suppose also that the k most highly ranked items (articles) w.r.t. a relevant query arrive in the very beginning, before all others. Using window semantics, the query will receive these k items and the result set will temporarily converge. The query will only receive the more recent articles with potentially fresh information much later, when the previous k items expire. Decay functions, on the other hand, consider time as part of the ranking function: in contexts like the news, where more recent information should be considered as more important, decay functions provide a tunable trade off between the importance of the content of an item and the importance of its recency.

Of course the price to pay for this continuous scoring is the computational cost



(a) Time-based sliding window

(b) Exponential decay

Figure 2.4: The score of three items with respect to a single query over time. The red lines shows the top-2 items at every time instant.

of updating frequently all scores and thus decay functions have been avoided when processing of continuous queries. Our work in [VAC12] was the first algorithm proposed over streaming information, incorporating decay functions instead of data windows, in the continuous top-k ranking model in an efficient way.

2.2 Scoring functions over text streams

In this section and in the rest of this chapter we focus on the continuous top-k ranking model over text streams. As discussed earlier, in the case of continuous top-k queries, the underling scoring function is of very high importance, since it determines not only the quality of results, but also the complexity of solving the query evaluation problem. In the following we present a number of scoring function parameters proposed in the literature, mainly in the context of online media and real-time streams. These can be classified in three main categories:

- *query dependent parameters* ($\mathcal{S}_{qu}(q, i)$), like text similarity functions, which determine the relevance of a given query w.r.t. a given item. Such parameters guarantee that results sent are relevant to the submitted queries.

- *static, query independent parameters* ($\mathcal{S}_{sta}(i)$), like source authority or novelty of information. Such parameters estimate the overall importance of a given item independently to any single query or user. For instance, the authority of the

source can be estimated given the PageRank [PBMW99] of the website or the number of past views on this source.

- *dynamic, query independent parameters* ($\mathcal{S}_{dyn}(i)$), like user feedback on a given item. Similarly to the previous category, these parameters are also query-independent, but dynamically alter the score of an item over time. For instance, the feedback received on a given item can increase or decrease its score and thus, provoke its insertion or deletion at some time after the item's arrival.

Generally, in the literature these three categories are linearly combined to compute a total score ($\mathcal{S}_{tot}(q, i)$) between a query (q) and an item (i):

$$\mathcal{S}_{tot}(q, i) = \alpha \cdot \mathcal{S}_{qu}(q, i) + \beta \cdot \mathcal{S}_{sta}(i) + \gamma \cdot \mathcal{S}_{dyn}(i) \tag{2.1}$$

The coefficients α, β and γ are determined based on the system's requirements.

A decay function is usually applied on the resulting value in order to consider freshness of information (see Subsection 2.1.5).

2.2.1 Text similarity

When ranking an item w.r.t a query, their textual similarity is undoubtedly a very important factor as it represents a straightforward measure of relevance of their contents. To model this query-dependant parameter, in majority of systems, both queries and items are represented using the *Vector Space Model (VSM)*: each term t of a query q (item i) is assigned a weight $w_{q,t}$ ($w_{i,t}$) and the query (item) is represented as a vector of these weights. Many weighting schemes have been proposed, with *tf-idf* term weighting being one of the most common ones. Based on the VSM, the main similarity functions commonly used are *cosine similarity*:

$$cos(q, i) = \frac{\sum_{t \in q \cap i} w_{q,t} \cdot w_{i,t}}{\sqrt{\sum_{t \in q} w_{q,t}^2} \cdot \sqrt{\sum_{t \in i} w_{i,t}^2}} \tag{2.2}$$

which is usually formulated in a simplified version (supposing appropriate normal-

ization of term weights):

$$cos(q,i) = \sum_{t \in q \cap i} w_{q,t} \cdot w_{i,t} \tag{2.3}$$

and *Okapi BM25* [RWJ+95]:

$$bm25(q,i) = \sum_{t \in q \cap i} w_{q,t} \cdot idf(t) \cdot \frac{w_{i,t}(k_1+1)}{w_{i,t} + k_1 \cdot (1 - k_2 + k_2 \cdot \frac{|i|}{avg_{i' \in I}(|i'|)})} \tag{2.4}$$

Even though text similarity captures the relevance between a query and an item, the use of a cosine or Okapi BM25 function alone is not sufficient for an effective filtering of the incoming streams. The following parameters proposed in the literature aim at estimating the importance of an item, independently to any query.

2.2.2 Item and item's source authority

In every modern search engine, the so-called authority of a web page plays a very important role in ranking documents. The measure used to capture this measure is usually PageRank [PBMW99] and it relies on hyperlinks towards a given web page to estimate its authority.

Techniques similar to PageRank have been proposed in the context of news, to measure the authority of a given item [DCGR05, HLLM06, MC10, MLY+10b]. Since it is rather infrequent to have links towards newly published information, these works create a graph of virtual links, which are in fact similarity connections between items. The proposed solutions also consider the temporal dimension of news. [MC10] bases its ranking on the T-Rank algorithm [BVW04], a variation of PageRank considering time.

The source authority is also considered as an important ranking parameter, especially in works over social network. [CLOW11] proposed the use of the social network's underlying user graph to estimate the PageRank of a user (represented by a node in this graph). This authority estimation is then used to assign an initial score of importance on the information the user publishes.

Depending on their definition, these measures can be either static or dynamic query-independent factors: they are considered as static if they are assigned to the item at the time of its arrival and never updated in any future time instant.

2.2.3 Media focus

In the context of news streams, in order to decide on the importance of an article, it is crucial to determine the importance of the real-world event it refers to. To do that, several studies propose clustering of articles in stories, i.e. sets of articles referring to the same real-world event. The more articles are added the a story, the more important the story is considered [DCGR05, MC10, WZRM08]. The intuition of this measure is that if the real-world event at which the articles refer to is important, then there should be a large number of sources referring to it. Once more, depending on its definition, this measure can be considered as a dynamic or static query-independent parameter.

2.2.4 Results' diversity

Also in the context of news, *diversity* of information has also been pointed out as an important factor when ranking items [DSP09, AAYI+13, AK11, MSN11]. In some cases, and especially when important real-world events occur, many articles may be published on the same information. In this case, it is essential to send to users articles with some diversity, i.e. articles that contain additional information w.r.t. what the user has already received.

2.2.5 User attention and feedback signals

Another important factor in ranking information deriving from streams, for both news streams and real-time streams, is the feedback received by other users. Such indications of *users' attention* include explicit feedback, with the use of a "like" button, sharing or commenting on the information, or implicit feedback, such as the number of page views. In the context of news streams, [WZRM08] proposes assigning higher scores to items read by more users. [DDGR07] also proposes using users click behavior over the Google News[12] commercial news aggregator in order to measure the importance of an article modeling the problem as a recommender system.

[12] news.google.com

2.2.6 Freshness of information

An important aspect when considering streaming information is the temporal dimension. As discussed in Section 2.1 either sliding windows or decay functions are usually employed over streaming information. Although there are a few works over text streams based on sliding windows [LDP10], decay functions have been more broadly used. Decay functions commonly employed include exponential [DCGR05, VAC12, SGFJ13], linear, polynomial [CLOW11] and sigmoid [WZRM08] decay.

2.3 Indexing and query evaluation algorithms

Based on the continuous top-k ranking model and the scoring functions discussed previously, in this section we present the most important algorithms and data structures proposed so far in the literature. We make the distinction between the two main approaches, namely the continuous approach, presented in Subsection 2.3.2 and the periodic execution of snapshot queries approach, presented in Subsection 2.3.3.

2.3.1 Preliminaries

The majority of algorithms on filtering text information rely on the baseline structure of *dictionaries* and on the Threshold Algorithm [FLN03]. Before explaining the most important algorithms proposed using either the continuous or the periodic snapshot queries approach, we explain in the following these baseline notions.

Dictionaries

A *dictionary*, a.k.a. *inverted index* (Figure 2.5a) is the most commonly used baseline data structure for the filtering of textual information. They represent a mapping from a term (word) to its *posting list*, i.e. the set of elements that contain it. When filtering queries over a set of documents, a dictionary stores for each term, the corresponding set of documents. In the case of continuous top-k queries, where documents (items) are evaluated over stored queries, the posting lists contain the set of queries with a given term. In a straightforward, naïve implementation for the continuous top-k query evaluation problem, queries would be indexed on all their terms and incoming items

would be evaluated by checking all queries in the posting lists corresponding to the item's terms.

Threshold Algorithm (TA)

TA [FLN03] was invented in 2003[13] and its goal is to efficiently retrieve the top-k documents for any given snapshot query. It supposes the use of the Vector Space Model to represent queries and documents and the use of cosine similarity as the underlying scoring function for query-item pairs (Equation 2.3). Documents are indexed in a dictionary, mapping each term to the set of elements containing it. Instead of being unordered sets, the posting lists of the dictionary are sorted based on the descending order of the documents' term weights. This order allows the definition of early stopping conditions when retrieving the top-k list. Every time a query is submitted, the posting lists corresponding to its terms are retrieved and only *partially* scanned to find the k best matching results. The order in the posting lists and the stopping conditions defined, guarantee that there will be no false negatives, i.e. the resulting top-k list of the query will be correct.

This algorithm presents the basis for numerous works proposed in the field of Information Retrieval and of Databases. Majority of the state of the art solutions on the continuous top-k query evaluation problem are also adaptations of the TA.

2.3.2 Continuous top-k query evaluation over text streams

SIFT [YGM99] was the first system proposed in the literature evaluating continuous queries over text streams. Although filtering was based on textual similarity between queries and items, the result was not based on the actual ranking. Several years later, the *Incremental Threshold* [MP11] and the *COL-Filter* [HMA10] were the first two systems to propose algorithms and data structures for the evaluation of continuous queries applying the continuous top-k ranking model. Both these systems considered the cosine similarity scoring function without being able to capture any static or dynamic query-independent scoring signals. Our work in [VAC12] was the first one to support more generalized scoring functions, capturing both textual similarity of query-item pairs and

[13] TA was in fact, an optimized version of an earlier algorithm of Fagin [Fag02]

query independent parameters. Additionally, this work was the first one to introduce decay functions, instead of data windows to capture the temporal aspect of the stream. The more recent works in [SGFJ13] and [RCCT14] also considered only textual similarity for the ranking of items w.r.t. queries. Figure 2.5 shows an abstract overview over the aforementioned systems. Starting from the baseline idea of *dictionaries*, we analyze in the following the aforementioned systems and their limitations.

SIFT

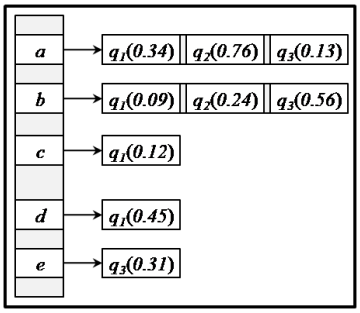
SIFT [YGM99] (Figure 2.5b) modelled the continuous query evaluation problem in the following way: a static threshold was set per query. Using the Vector Space Model to represent queries and items, and cosine similarity as the scoring function, an arriving item would be inserted in a query's result set if the similarity score exceeded the query's threshold. In the proposed solution, a query would be indexed in the dictionary on only a subset of its terms, called "significant", that had a high term weight. The intuition was that if an item contained only non-indexed terms, their weights would not be sufficient for the document to update them. In any other case, the document would be found in the posting of an indexed term and sent to the query.

The use of this optimization was valid in the case of a static threshold, however, in the continuous top-k ranking model it cannot be applied: the threshold of each query is the minimum score of its last (k-th) element and is, thus dynamic as well as the list of the potentially significant terms. Another limitation of the SIFT system is that all queries in the posting lists of each item's terms need to be checked for updates, without using any early stopping condition.

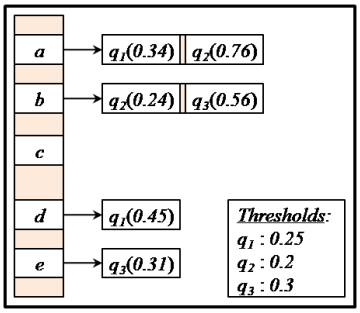
Incremental Threshold

The *Incremental Threshold* algorithm [MP11] (Figure 2.5c) was the first one to use the continuous top-k ranking model over text streams and it considered sliding window semantics. In the proposed solution two dictionaries are maintained, one over the queries and one over the items. The item's dictionary only contain the most recent items, i.e. the ones that belong in the data window: on arrival, an item is inserted in the items' dictionary, while on expiration it is removed. The posting list of any term is

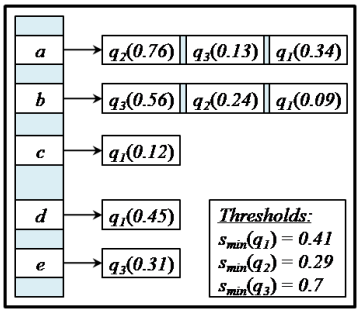
$$q_1 = \{a(0.34), b(0.09), c(0.12), d(0.45)\}$$
$$q_2 = \{a(0.76), b(0.24)\}$$
$$q_3 = \{a(0.13), b(0.56), e(0.31)\}$$



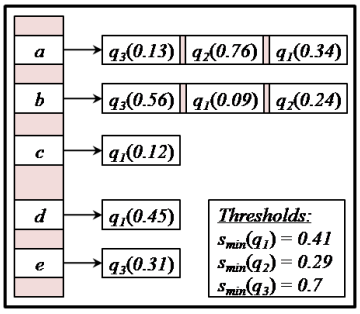
(a) A term to queries dictionary



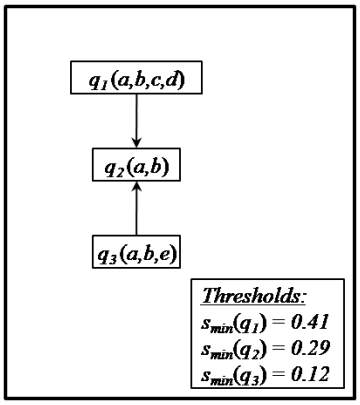
(b) SIFT [YGM99]



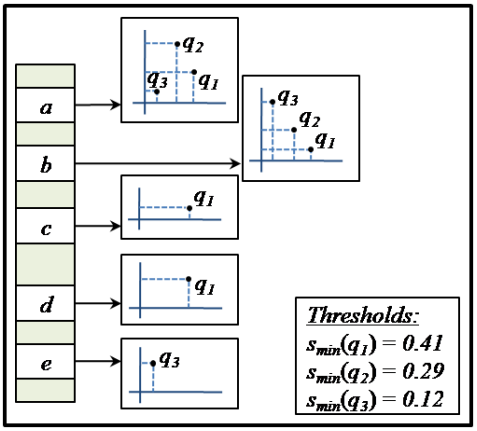
(c) Incremental Threshold [MP11]



(d) COL-Filter [HMA10]



(e) GIS [RCCT14]



(f) Our proposed solution [VAC12] presented in Chapter 3

Figure 2.5: Overview of a simple dictionary (a) and systems proposed for the continuous top-k query evaluation problem (b-f).

sorted in descending order by the term weight assigned to the corresponding item and for the given term, exactly like in the Fagin's *Threshold Algorithm (TA)*. In the second dictionary, built over the stored continuous queries, the posting lists are sorted by a value $\theta_{q,t}$, where t is the corresponding term of the posting list and q is a query that contains it.

At query insertion, for the initial computation of the top-k list, the first data structure is used, by directly applying TA. Recall that TA algorithm, based on the ordering of the postings, can guarantee an early stopping condition while iterating through the lists, so that not all entries need to be scanned. Once the top-k list of the new query is calculated, the $\theta_{q,t}$ values are initialized and their value indicates the minimum document term weight for term t that could influence the top-k result of the given query.

On arrival, each item is inserted in the items' dictionary. Then, using the queries dictionary, the updated queries are retrieved: Knowing the term weight for each term in the item, we retrieve from the posting list of queries, those where the value of $\theta_{q,t}$ is lower than the items term weight, as according to q,t definition, these are the ones that could be updated by the item. This condition guarantees that there will be no false negatives.

One main drawback of this solution is the need for continuous updates on the inverted index of items on item publications and expirations, which create a high system overload. Due to this, and despite the definition of an early stopping condition, experimental evaluation in [HMA10] shows that the Incremental Threshold solution requires up to 60% more time than the naïve solution on item evaluation.

COL-Filter

The *COL-Filter* algorithm (Figure 2.5d), proposed in [HMA10], is the first solution achieving a significantly better performance, in terms of item evaluation time, compared to the naïve solution. Like the Incremental Threshold algorithm, it assumes sliding windows and also uses a variation of TA to retrieve the set of updated queries. In COL-Filter a single dictionary is maintained over stored queries. In the posting list of each term t, all queries q are sorted in descending order on a value $w_{q,t}/\mathcal{S}_{min}(q)$, where $w_{q,t}$ is the term wait of a query q for an item i and $\mathcal{S}_{min}(q)$ is the score of the k-th item

in the top-k list of q. Notice that the value $\mathcal{S}_{min}(q)$ sets a threshold score for an item to be inserted in the query's result set.

At query insertion, the query has an zero minimum score $\mathcal{S}_{min}(q)$ so, the query is always inserted in the beginning of the posting list. On query update, i.e. when the value $\mathcal{S}_{min}(q)$ is changed, the query is re-indexed with the posting list. On item matching, the posting lists of all terms of the arriving item are retrieved. Then, in a round-robin way the queries in the lists are visited and checked for potential updates. The authors in [HMA10] have proven a stopping condition over this iteration, guaranteeing that after a given condition, it is safe to stop the algorithm, without any false negatives, i.e. without missing any queries that should be updated.

The main limitation of both COL-Filter and Incremental Threshold is that the assumed scoring function considers only textual similarity between the query and the item, and cannot include query-independent item scores. More precisely the Incremental Threshold algorithm assumes cosine similarity, while COL-Filters assumes a more general form using a combination of monotonic, homogeneous functions. As defined in [HMA10] a function $f(x_1, ..., x_m)$ is monotonic *iff* $f(x_1, ..., x_m) \leq f(x'_1, ..., x'_m)$ when $x_j \leq x'_j$ for every value of j. A function is *homogeneous* if it preserves the scalar multiplication operation: $f(\alpha x_1, ..., \alpha x_m) \leq alpha^r f(x_1, ..., x_m)$. Consequently, functions such as cosine similarity (Equation 2.3) or Okapi-BM25 (Equation 2.4) can be used. However, these only correspond to the query dependent part of the scoring function and cannot support query-independent parameters (see Section 2.2).

GIS

In the GIS algorithm [RCCT14] (Figure 2.5e), a graph connecting continuous queries is the principal data structure stored by the system. A connection from one query to another indicates a subset relationship between their term sets. Using these semantics, an initial graph of queries is created when *all* continuous queries are stored in the system. A simplified directed acyclic graph (DAG) is then created by removing some edges. On item arrival, this graph is traversed from root-nodes towards the leaves, while under a number of conditions the algorithm can terminate early.

A key disadvantage of the GIS algorithm, is that it employs a simplified term weighting scheme where all query weights are equal to 1. Furthermore, it works under the

assumption that no query insertions or deletions can be performed after the initial graph has been computed, making this algorithm inadequate for modern, highly dynamic applications.

2.3.3 Real-time search over Twitter

The so-called *real-time web* has received significant commercial and research attention over the past few years. While Twitter remains today the most popular way of publishing real-time information, major search engines like Google[14] and Bing[15], having signed appropriate deals, have complete access on Twitter's streams and provide real-time results, i.e. recent tweets, for queries they receive on trending topics.

In the literature, Twitter Index [CLOW11] was the first real-time search algorithm proposed over the Twitter's stream. A year later, [BGL+12] presented some insights on how Twitter's Search actually works. Several works that followed [WLXX13, MME+14, LBLT15] also focus on providing efficient real-time search. However, none of the proposed algorithms relies on a continuous top-k evaluation approach and only focus on the snapshot query scenario. As explained in Chapter 1 the goal of these solutions is to efficiently index incoming information items (tweets) and they define real-time search as "*the ability to ingest content rapidly and make it searchable immediately, while concurrently supporting low-latency, high throughput query evaluation.*" [BGL+12]. With this approach corresponding to the functionality of the query handler component (see Figure 1.3), long-running queries can only be supported by periodically executing the equivalent snapshot queries. Twitter's Search applies this approach and re-retrieves the top-20 items every 30 seconds. The drawbacks of this approach w.r.t. continuous queries have been discussed in Subsection 2.1.4.

Although through our work, we do not focus on the query handler component and the support of (periodic) snapshot queries, in the following we present *Twiiter Index (TI)* [CLOW11] and *Earlybird* [BGL+12], the most important works in the context of real-time search.

[14] searchengineland.com/google-twitter-deal-live-221148

[15] blogs.bing.com/search/2011/03/25/bing-feature-update-bing-news-with-real-time-twitter-feed-and-enhanced-entertainment-sharing/

TI: Twitter index.

TI [CLOW11] was the first work focusing on the efficient indexing of real-time web content published on Twitter, in order to make it immediately available through user snapshot queries. For each arriving query, the aim is to retrieve the top-k items (tweets). A scoring function similar to the one defined in Equation 2.1 is employed, using a linear combination of textual similarity and both static and dynamic query-independent parameters. Cosine similarity is used to estimate the textual similarity between queries and items. "*User PageRank*", a measure capturing the authority of the user, represents the static estimation of the item's importance. Replies on tweets are considered as positive feedback events and increase the tweet's dynamic query-independent score. Finally, a rational decay function is applied on the result to capture the freshness of information ($\mathcal{S}_{tot}(q,i)/\Delta\tau$, where $\mathcal{S}_{tot}(q,i)$ is the initial query-item score and $\Delta\tau$ is the time difference since the publication of the item).

The TI algorithm maintains both in-memory and secondary storage data structures in order to deal with the huge memory requirements for storing data deriving from a stream (due to decay, old tweets can eventually be removed). To optimize the performance of their overall system, they distinguish between the *noisy* and the *significant* tweets. The tweets classified as noisy, are considered as less likely to appear on any future query result set and indexed using a slower, batch processing approach. Significant tweets, on the other hand, are immediately indexed.

To answer incoming snapshot queries, an inverted index of recent tweets is maintained and its posting lists are ordered based on the arrival time of the tweet. This order permits the definition of an early stopping condition on query evaluation: supposing that while traversing the posting lists of the query's terms, k items have been found with a query-item score greater than a value s_{min}, the scanning of the posting lists can stop, *iff* s_{min} is greater than the maximal value any tweet can have given that it is published at the exact same time as the one currently being checked. This maximal value is given based on the score decay function. Due to ordering of tweets imposed by the algorithm, it is easy to prove that the stopping condition is correct and that the top-k list of tweets will be accurate for any given query.

Earlybird

Earlybird [BGL+12] is a work published by Twitter and presents the actual system's architecture, along with an abstract description on how Twitter's Search actually works. There, also, the main components of the scoring function are text similarity, query-independent (static and dynamic) item importance parameters and information freshness (using a decay function). Additionally, they also consider other user-related parameters, such as their *followers* or *followees*.

Like in TI, the posting lists of the inverted tweets index maintained by Earlybird are sorted in chronological order so as to easily access recent tweets. To achieve efficient indexing of incoming tweets, Earlybird used compression techniques for tweets' contents and for long posting lists, multithreaded algorithms and resources allocation optimizations. Although the query evaluation algorithm is not explained in detail, the authors say that a variation of Lucene's algorithm[16] is used.

[16] lucene.apache.org/core/

3

Query Filtering over Text Streams

In this chapter we are interested in the scalable processing of content filtering queries over text item streams. In particular, we aim at generalizing state of the art solutions with scoring functions combining query-independent item importance with query-dependent content relevance. While such complex ranking functions are widely used in web search engines this is the first scientific work studying their usage in a continuous query scenario. Our main contribution consists in the definition and the evaluation of new efficient in-memory data structures for indexing continuous top-k queries based on an original two-dimensional representation of text queries. We are exploring locally-optimal score bounds and heuristics that efficiently prune the search space of candidate top-k query results which have to be updated at the arrival of new items. Finally, we experimentally evaluate the memory and matching time trade-offs of these index structures. In particular we experimentally illustrate their linear scaling behavior with respect to the number of indexed queries.

With respect to our proposed functionality presented in Chapter 1 (see Figure 1.3), in this chapter we focus on the Item Handler component as shown in Figure 3.1: for each item, we perform the matching operation immediately as it arrives from the stream, against stored queries. We suppose that the query-independent item score is static and thus, there are no feedback signals on items. Chapter 4 describes our approach in dealing with dynamic item scores and presents our solutions for implementing the Event Handler component.

The rest of the chapter is organized as follows. Section 3.1 gives a formal definition of the problem. Section 3.2 presents our proposed inverted query indexing and pruning techniques. Then, Section 3.3 describes a number of index implementations that apply these techniques. In Section 3.4 we present a probabilistic model for the definition of an approximate solution with maximum error guarantees. Related work is presented in Section 3.5 and Section 3.6 provides the experimental evaluation. Finally, Section 3.7 summarizes the main contributions of this work.

The results of the work presented in this chapter have been published in [VAC12]. The source code of all the implementations is publicly available as an open source

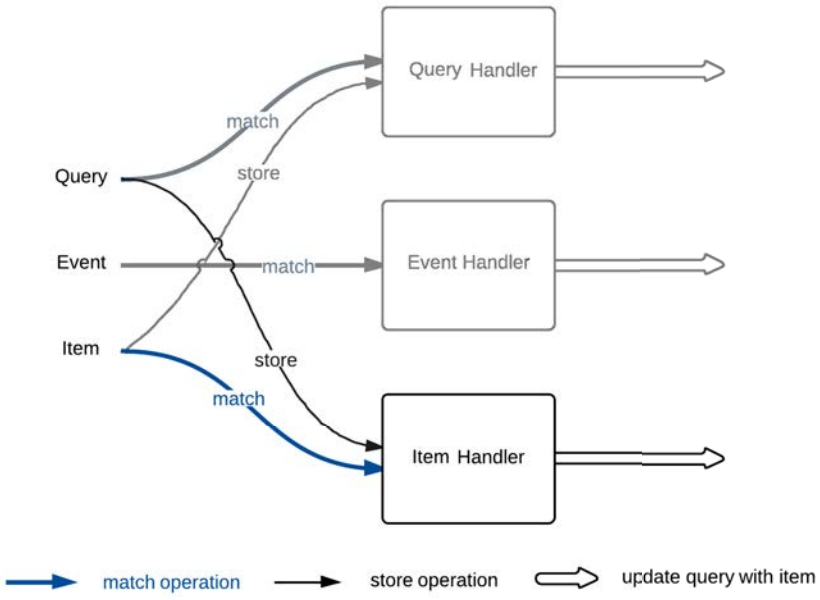


Figure 3.1: Focusing on the Item Handler component. Submitted queries are stored in the item handler. Incoming items are matched against them and potential updates in the query's top-k lists are retrieved

project[1].

3.1 Problem statement

This section presents a formal definition of the problem of continuous top-k queries combining query-dependent and query-independent scores for ranking items from textual data streams.

3.1.1 Queries, items and scores

We follow the traditional Vector Space Model approach [MRS08] for the definition of items, queries and content similarity.

Definition 1 (queries and items)**.** *A vocabulary of terms V defines a $|V|$-dimensional space $\mathbb{R}^{|V|}$ where each dimension corresponds to a term t in V. Each vector $\vec{\omega}$ in $\mathbb{R}^{|V|}$*

[1] code.google.com/p/continuous-top-k/

represents either a query *q or a text* item *i over V. We denote by* $\omega_{q,t}$ *and* $\omega_{i,t}$ *the* weight *of term t in query (vector) q and item (vector) i respectively.*

Term weights can be assigned to queries and items using any weighting scheme, such as the tf-idf scheme [SB88]. Without loss of generality, we assume that query and item weights are both normalized, i.e. the sum of the weights for each query and each item is equal to 1. Finally, we say that a query q or an item i is relevant (irrelevant) to some term t if the corresponding weight $\omega_{q,t}$ or $\omega_{i,t}$ respectively, is greater than (equal to) 0. This definition allows us to use a simplified set semantics for terms and items and to write $t \in q$ or $t \in i$ to denote that t is relevant to q or i respectively.

Our goal is to rank items with respect to queries by combining *query-dependent* similarity scores with *query-independent item scores* [ZSYW10] (see discussion in Section 2.2:

Definition 2 (total scores)**.** *Let Q be a possibly infinite set of queries and I be a possibly infinite set of items. We define three ranking functions:*

- $\mathcal{S}_{qu} : Q \times I \to [0, 1]$ *is a query-dependent score representing the textual similarity between a query q and an item i, called* query score *of q and i. We present our approach using* cosine *similarity:*

$$\mathcal{S}_{qu}(q, i) = \sum_{t \in V} \omega_{q,t} \cdot \omega_{i,t}$$

  However, more complex score functions like Okapi BM25 could also be applied. Our approach essentially works for any kind of vector-based similarity scores applying a monotonic *aggregation function (sum) over a set of scores obtained by a monotonic weight combination function (multiplication).*

- $\mathcal{S}_{ite} : I \to [0, 1]$ *is a static, query-independent score called* item score *of* $i \in I$*. It can reflect any combination of ranking parameters like novelty of information [GDH04], source authority [DCGR05, HLLM06, MC10, MLY+10a] or user attention [WZRM08, LDP10]. However, it cannot reflect any dynamic parameters, such user attention (see Section 2.2).*

- $\mathcal{S}_{tot} : Q \times I \rightarrow [0,1]$ *is called* total score *of* q *and* i *and is defined as the weighted sum of* $\mathcal{S}_{qu}(q,i)$ *and* $\mathcal{S}_{ite}(i)$*:*

$$\mathcal{S}_{tot}(i,q) = \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot \mathcal{S}_{qu}(q,i) \tag{3.1}$$

where α *is a non-negative constant in the interval* $[0,1]$ *and* $\beta = 1 - \alpha$*. The aforementioned score values constraints guarantee that* $\mathcal{S}_{tot}(i,q) \in [0,1]$*.*

3.1.2 Item streams and score decay

So far, we have considered a possibly infinite set of items I. This set is published in form of a stream $I(\tau)$ of items published until time instant τ: $I(\tau) \subseteq I(\tau') \subseteq I$ for all time instants $\tau < \tau'$. Let Q be a finite set of queries and $\mathcal{S}_{tot} : Q \times I(\tau) \rightarrow [0,1]$ be a ranking function which computes for each item $i \in I(\tau)$ and query $q \in Q$ a *static total score* $\mathcal{S}_{tot}(i,q)$[2].

Freshness of information is an important aspect for ranking items deriving from a stream. In most applications publishing content streams, it is natural to appreciate more recently published pieces of information as more important than older ones. The two strategies most commonly employed over streams in order to guarantee freshness of information are *sliding data windows* and *aging models* which employ *time decay functions*. While sliding windows has been a methodology widely used for ranking tuple based streams [LYWL05, TP06, MBP06, PZA08], more recent approaches on ranking text streams propose using aging models which continuously decay scores instead (see Subsection 2.1.5). We also follow the aging models approach in this work.

Definition 3 (decay function)**.** *A function* $decay : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ *is a* function *applied on a score* $s \in \mathbb{R}$ *after some time interval* $\Delta\tau \in \mathbb{R}$ *iff* $decay(s,0) = s$ *and* $decay(s,\Delta\tau)$ *is monotonically decreasing for increasing* $\Delta\tau$*:*

$$\Delta\tau \geq \Delta\tau' \Rightarrow decay(s,\Delta\tau) \leq decay(s,\Delta\tau')$$

[2]Observe that $\mathcal{S}_{tot}(i,q)$ is undefined until the publication of item i in stream $I(\tau)$ and remains constant after its initial definition.

Definition 4 (decayed score)**.** *Let constant τ_i denote the publication date of item i and τ be some time instant after τ_i. The* decayed score $\mathcal{S}(i, q, \tau)$ *of item i with respect to q at some time instant τ is then defined as the decay of the total score $\mathcal{S}_{tot}(i, q)$ with respect to the age $\tau - \tau_i$ of item i:*

$$\mathcal{S}(i, q, \tau) = decay(\mathcal{S}_{tot}(i, q), \tau - \tau_i) \tag{3.2}$$

3.1.3 Continuous top-k queries

Continuous top-k queries and results over a stream are defined as follows.

Definition 5 (top-k query and result)**.** *The top-k result $R(q, \tau, k)$ of some (top-k) query $q \in Q$ at some time instant τ with respect to some decayed ranking function $\mathcal{S}(i, q, \tau)$ is an ordered subset of maximally k items $i \in I(\tau)$ such that i shares at least one term with query q and there exists no item $i' \in (I(\tau) - R(q, \tau, k))$ where $\mathcal{S}(i', q, \tau) > \mathcal{S}(i, q, \tau)$.*

Less formally, $R(q, \tau, k)$ contains at each time instant τ the subset of the k most relevant items for a query q. Based on the previous definition we can state the following general continuous top-k query evaluation problem:

Problem 1 (continuous top-k query evaluation)**.** *Given a set of queries Q, a decayed ranking function $\mathcal{S}$ and an item stream $I(\tau)$, maintain for each query $q \in Q$ its top-k result $R(q, \tau, k)$ at any time instant τ.*

At a processing level this general problem is solved by a system which adds each new stream item i in all relevant top-k results $R(q, \tau_i, k)$ *and* maintains the results by removing and replacing items due to score changes on scores over time. These two problems can be defined separately.

The first problem can be solved by a transactional system where each item arrival triggers a transaction maintaining the top-k lists of *all relevant* queries in Q. Updates are atomic actions isolated from each other.

Sub-problem 1 (query filtering problem)**.** *We denote by $\mathcal{S}_{min}(q, \tau)$ the score of the last item in $R(q, i, k)$ at time instant τ. Given a set of queries Q and an item i arriving*

at some time instant τ_i, update the top-k result for all queries $q \in Q$ where

$$\mathcal{S}_{tot}(i, q) > \mathcal{S}_{min}(q, \tau_i) \tag{3.3}$$

Observe that decay does not need to be applied on the score of the newly arriving item, as the time interval since its publication is of zero length. We will denote by $U(i)$ the set of all queries whose top-k result will be updated by item i at the arrival time instant τ_i:

$$U(i) = \{q | \mathcal{S}_{tot}(q, i) \geq \mathcal{S}_{min}(q, \tau_i)\} \tag{3.4}$$

It is easy to show that the complexity of the query filtering problem depends on the number of queries $|Q|$ and the ranking function $\mathcal{S}_{tot}$. A trivial solution to this problem is to compute at the arrival of each new item i at some time instant τ_i its score $\mathcal{S}_{tot}(i, q)$ and compare it to the minimal score $\mathcal{S}_{min}(q, \tau_i)$ of all queries which share at least one term with the item. However, this solution obviously does not scale in the size of the items, i.e. number of terms it contains, and in the number of queries relevant to these terms. A better solution then is to define appropriate index structures that prune the search space and avoid searching all queries $q \in U(i)$ for potential updates. In this work we are concerned by the increased complexity of the filtering problem introduced by non-homogeneous ranking functions (see subsection 3.1.1). This is to our knowledge the first scientific work in this direction.

The second sub-problem derives from the application of decay on the scores, which adds a temporal dimension to the query filtering results: as time goes by, all scores change due to decay and even without the arrival of new items, top-k results can potentially change. We will call this the *result maintenance problem.*

Sub-problem 2 (result maintenance problem)**.** *Given a decayed ranking function $\mathcal{S}(i, q, \tau)$, maintain for each query $q \in Q$ its top-k result $R(q, \tau, k)$ for any time instant τ.*

The complexity of result maintenance problem obviously depends on the *decay* function. We can essentially distinguish between two kinds of decay functions:

1. *Order-preserving* functions, which guarantee that the relative position between two scores at any time instant τ remains the same at any other time instance t'

in the future or the past.

2. *Non order-preserving* decay functions, which do not fulfill the previous property.

More formally, order preserving decay guarantees that for all time instants τ, τ', all items i, i' and all queries q, if $\mathcal{S}(i,q,\tau) < \mathcal{S}(i',q,\tau)$ then $\mathcal{S}(i,q,\tau') < \mathcal{S}(i',q,\tau')$. In this case, the continuous top-k query problem is limited to query filtering, as top-k results can only be altered by the arrival of new items. In the second non order-preserving case the top-k results might change independently to the arrival of new items, making the result maintenance problem much more difficult. It is worth noticing that most of the recent works on streams of news or blogs consider linear [WZRM08] and exponential decay functions [DCGR05, MC10], which are order-preserving , rather than the opposite as in the case of Sigmoid functions[HLLM06]. Note also that sliding window could be defined as a non order-preserving decay function. In this work we do not consider the problem of result maintenance generated by non-order preserving decay functions.

Homogeneous versus non-homogeneous ranking functions Ranking functions employed by web search engines [ZSYW10] are in general more complex than those considered by existing continuous top-k query evaluation algorithms [MP09, HMA10, MP11, HMA12]. They essentially combine different query-dependent and query-independent scores which go beyond the *monotonic* and *homogeneous* text similarity scores like *cosine* similarity. More precisely, a ranking function $sim : \mathbb{R}^{|V|} \times \mathbb{R}^{|V|} \to \mathbb{R}$ over some vector space $\mathbb{R}^{|V|}$ is homogeneous of degree i if $sim(n \cdot x, n \cdot y) = n^i sim(x, y)$ for all non-zero $n \in \mathbb{R}$ and vectors $x, y \in \mathbb{R}^{|V|}$. It is easy to see that this property holds for functions like *cosine* similarity, but not for our definition of total score with non-zero α. Homogeneity and monotonicity are two necessary conditions for existing threshold-based algorithms for defining a total order over items (snapshot query setting) or queries (continuous query setting). Contrary to these works, we are interested in continuous queries featuring non-homogeneous ranking functions. As we will see in the following section, this extension raises new challenges to the query filtering problem.

3.2 Query filtering

In this section we present a new approach for processing large collections of continuous top-k queries over item streams. From a processing point of view the query filtering problem consists in identifying for each new item i all queries $U(i)$ which must add item i to their top-k result. The main optimization goal is to reduce this search space and to compute a smallest possible number of *candidate queries* containing $U(i)$. Our solution works for *non-homogeneous* ranking functions which makes it more general than other continuous top-k query filtering solutions that are restricted to *monotonic* and *homogeneous* functions for computing item scores [MP11, HMA10, HMA12]. This generalization is achieved by replacing a total ordering of queries with a two-dimensional query representation. Based on this search space representation, we then introduce a number of linear constraints spatially characterizing different sets of candidate queries. Later, in Section 3.3, we present and compare a different index implementations for efficiently evaluating these conditions.

3.2.1 Query filtering without decay

For the sake of simplicity, we first abstract the notion of time and decay by considering that all scores are computed for a fixed time instant τ_0 (all variables referring to time instants or time periods disappear from the corresponding definitions). Later, in Section 3.2.2 we will show how decay can be added to this scenario.

For each term t in the vocabulary of terms V, we define a set of points in a two-dimensional space called the *inverted query index* of t and denoted $P(t)$:

Definition 6 (inverted query index)**.** *The inverted query index $P(t)$ of a term t over a set of queries Q is the set:*

$$P(t) = \{(\mathcal{S}_{min}(q), \omega_{q,t}) | q \in Q \wedge \omega_{q,t} > 0\} \tag{3.5}$$

where $\mathcal{S}_{min}(q) = \mathcal{S}_{min}(q, \tau_0)$.

Observe that each inverted query index $P(t)$ contains all queries that are relevant for term t. Equivalently, for finding all candidate queries of an item i it is sufficient to

explore the inverted query indexes $P(t)$ of all terms $t \in i$.

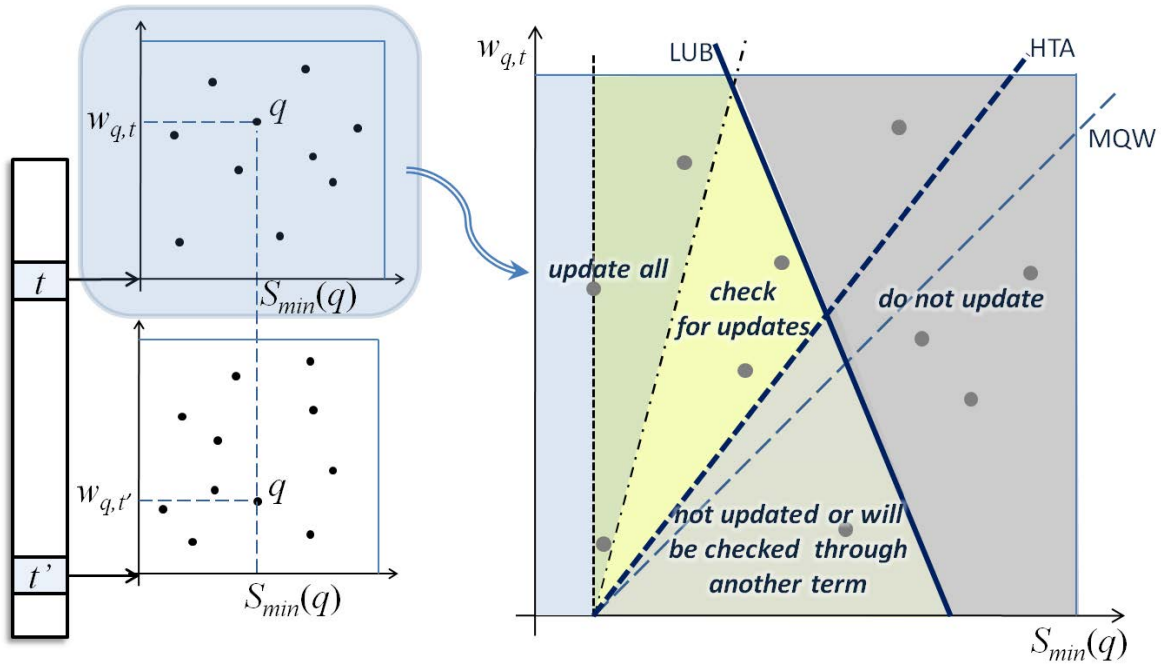


Figure 3.2: Query representation

For example, let q be a query containing two terms, t and t'. As we can see in Figure 3.2, q is stored in the corresponding indexes $P(t)$ and $P(t')$ and positioned in the coordinates defined by the term weights and minimum score at that time instant. The right part of the image zooms into $P(t)$ index. All queries q' in this index contain term t with a positive weight.

Starting from this search space representation we define three *linear upper bound conditions* filtering for each new item i and for any term $t \in i$ a subset of $P(t)$ containing *candidate* queries whose top-k result is potentially updated by i. For performance reasons, apart from knowledge on the item, each condition should depend only on information available *locally* for all queries q in the inverted query index $P(t)$ of a term t, i.e. the query term weight $\omega_{q,t}$ and the minimal top-k list score $\mathcal{S}_{min}(q)$. This locality reduces the precision of search space pruning conditions, but as other inverted file data structures it facilitates implementations that scale in the number of queries and the number of terms.

Local Upper Bound (LUB) In the following, we focus on the problem of bounding the query score $\mathcal{S}_{qu}(q,i)$. Obviously, it is impossible to compute the precise value of $\mathcal{S}_{qu}(q,i)$ for any query q without aggregating information from all query indexes $P(t)$ of terms t in i. However, we can estimate for each term t a value $M(q,i,t)$ which is an upper bound for the term-query weight product sum of the other terms in i as shown in the following equation:

$$\mathcal{S}_{qu}(q,i) = \sum_{t' \in V} \omega_{q,t'} \cdot \omega_{i,t'} = \omega_{q,t} \cdot \omega_{i,t} + \overbrace{\sum_{t' \in V, t \neq t'} \omega_{q,t'} \cdot \omega_{i,t'}}^{M(q,i,t)}$$

Let $\omega_{i,\bar{t}} = max\{\omega_{i,t'} | t' \in V \wedge t' \neq t\}$ be the maximum item weight of all terms t' in i different from t. Observe that $\omega_{i,\bar{t}} = 0$ for all items containing exactly one term (which is rather an exception). Then we can prove that $M(q,i,t) \leq \omega_{i,\bar{t}} \cdot (1 - \omega_{q,t})$ which leads to the following upper bound condition for $\mathcal{S}_{qu}(q,i)$:

$$\mathcal{S}_{qu}(q,i) \leq \omega_{q,t} \cdot \omega_{i,t} + \omega_{i,\bar{t}} \cdot (1 - \omega_{q,t}) \tag{3.6}$$

By replacing $\mathcal{S}_{qu}(q,i)$ in equation 3.1, we obtain the following first upper bound condition over $P(t)$ denoted LUB:

$$\mathsf{LUB} : \mathcal{S}_{min}(q) < \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot (\omega_{q,t} \cdot \omega_{i,t} + \omega_{i,\bar{t}} \cdot (1 - \omega_{q,t}))$$

Condition LUB defines a subset of queries $C_{\mathsf{LUB}}(i,t)$ of candidate queries in $P(t)$ as shown in figure 3.2. The only parameters are the item score $\mathcal{S}_{ite}(i)$ and both item weights $\omega_{i,t}$ and $\omega_{i,\bar{t}}$.

Theorem 1 (Correctness of **LUB**). *For all items i and all queries $q \in U(i)$, q appears in the candidate set $C_{\mathsf{LUB}}(i,t)$ of* all *terms t shared between q and i.*

Proof: *For proving the correctness of* LUB *(theorem 1) we can show that q appears in the candidate list of* all *terms t* shared by q and i.

Suppose that $q \in U(i)$ and t is a term shared by query q and item i. Let $\omega_{i,\bar{t}} = max\{\omega_{i,t'} | t' \neq t\}$ be the maximal weight of all terms different from t in item i(if i contains only term, then $\omega_{i,\bar{t}} = 0$).

Then, we only have to show that

$$\sum_{t' \in V, t \neq t'} \omega_{q,t'} \cdot \omega_{i,t'} \leq \omega_{i,\bar{t}} \cdot (1 - \omega_{q,t}) = M(q,i,t) \tag{3.7}$$

Since query term weights are normalized and $\omega_{i,\bar{t}} \geq \omega_{i,t'}$ *for all terms, we know that*

$$\sum_{t' \in V, t \neq t'} \omega_{q,t'} \cdot \omega_{i,t'} \leq \sum_{t' \in V, t \neq t'} \omega_{q,t'} \cdot \omega_{i,\bar{t}} \tag{3.8}$$

and

$$\sum_{t' \in V - \{t\}} \omega_{q,t'} = 1 - \omega_{q,t} \tag{3.9}$$

Inequality 3.7 directly follows from equation 3.8 and equation 3.9. □

From Theorem 1 directly follows that condition LUB is *safe*:

$$U(i) \subseteq \bigcup_{t \in V} C_{\mathsf{LUB}}(i,t)$$

Theorem 2 (Local Optimality of **LUB**). *If item* i *contains only one term* t *then* $C_{\mathsf{LUB}}(i,t)$=$U(i)$. *Otherwise, for each query* $q \in C_{\mathsf{LUB}}(i,t)$ *we can define a query* q' *which is* indistinguishable *from* q *in* $P(t)$ *(*$\omega_{q',t} = \omega_{q,t}$ *and* $\mathcal{S}_{min}(q') = \mathcal{S}_{min}(q)$*) such that* $q' \in U(i)$.

Proof: *We will prove that Condition* LUB *is locally optimal within the query index* $P(t)$ *of a given term* t *(theorem 2). To do that, we will show that* any *query* q *within the bounds of Condition* LUB *could potentially be updated. More formally, we will prove that for all queries* $q \in C_{\mathsf{LUB}}(i,t)$ *there might exist a query* $q' \in U(i)$ *which is* undistinguishable *from* q *in* $P(t)$ *(*$\omega_{q',t} = \omega_{q,t}$ *and* $\mathcal{S}_{min}(q') = \mathcal{S}_{min}(q)$*).*

Let q' be a query with two terms $\{t, \bar{t}\}$ *where*

$$\bar{t} = \arg\max_{t'} \{\omega_{i,t'} | t' \in V \wedge t' \neq t\}$$

Then, the total score of query q' is:

$$\begin{aligned}\mathcal{S}_{tot}(q', i) &= \alpha \cdot \mathcal{S}_{ite}(i) + \\ &\quad \beta \cdot (\omega_{q',t} \cdot \omega_{i,t} + \omega_{q',\bar{t}} \cdot \omega_{i,\bar{t}})\end{aligned}$$

By hypothesis, $\omega_{q',t} = \omega_{q,t}$ and since query term weights are normalized $\omega_{q',\bar{t}} = 1 - \omega_{q',t} = 1 - \omega_{q,t}$. Then we can rewrite the previous equation as follows:

$$\begin{aligned}\mathcal{S}_{tot}(q', i) &= \alpha \cdot \mathcal{S}_{ite}(i) + \\ &\quad \beta \cdot (\omega_{q,t} \cdot \omega_{i,t} + (1 - \omega_{q,t}) \cdot \omega_{i,\bar{t}})\end{aligned}$$

From the initial hypothesis that LUB *holds for query q and $\mathcal{S}_{min}(q) = \mathcal{S}_{min}(q')$ follows that $\mathcal{S}_{min}(q') < \mathcal{S}_{tot}(q', i)$, i.e. $q' \in U(i)$.* □

Theorem 1 states that each query $q \in U(i)$ appears as a candidate in $C_{\mathsf{LUB}}(i, t)$ of all terms shared between q and i. This introduces some redundancy in the query matching algorithm. In the following we define two additional conditions which can each be safely added to $C_{\mathsf{LUB}}(i, t)$ by conjunction to restrict the number of candidates for each term (however it is not possible to add both conditions without obtaining false negatives). The main idea is to exploit extra knowledge about the maximal query length and the query score distribution for still guaranteeing that each query candidate appears in the candidate set of *at least* one term.

Extreme Cases We can show that (1) LUB returns exactly the set of queries to be visited, i.e. $U(i) = C_{\mathsf{LUB}}(i, t)$ for items of length 1 and (2) all queries q of length 1 in $C_{\mathsf{LUB}}(i, t)$ have to be updated for any item i and any term t:

1. Let t be the only term of item i and $q \in C_{\mathsf{LUB}}(i, t)$. According to the definition of LUB, $\omega_{i,\bar{t}} = 0$ and we obtain the following condition:

   $$\mathcal{S}_{min}(q) < \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot (\omega_{q,t} \cdot \omega_{i,t})$$

   Since term t is the only term in the intersection of item i and query q, this last inequality immediately means that $q \in U(i)$

2. Since term weights are nomalized we know that $\omega_{q,t} = 1$ for all queries q of length 1. Thus, $(1 - \omega_{q,t}) = 0$ in equation 3.2.1 and we obtain the same upper bound condition and conclusion as before: $q \in U(i)$.

□

Higher Than Average (HTA) The first condition exploits the fact that for any query $q \in U(i)$, there exists at least one item/query weight product greater or equal to the average per term query score. This condition takes account of the *maximal number of terms a query can share with some item.* Since queries are in general shorter than items, this bound, denoted by λ, can be defined by the maximum query length. Then we define the following constraint on the query score for queries $q \in U(i)$ and for at least one term t:

$$\mathcal{S}_{qu}(q, i) \leq \lambda \cdot \omega_{q,t} \cdot \omega_{i,t} \tag{3.10}$$

and the corresponding condition:

$$\mathsf{HTA} : \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot \lambda \cdot \omega_{q,t} \cdot \omega_{i,t} > \mathcal{S}_{min}(q)$$

Condition HTA defines a subset of queries $C_{\mathsf{HTA}}(i, t)$ of candidate queries in $P(t)$ as shown in figure 3.2.

Theorem 3 (Correctness of **HTA**)**.** *For all queries q, all items i and all ranking functions $\mathcal{S}_{tot}(q, i)$, if $q \in U(i)$ then there exists at least one term t shared by q and i, such that $q \in C_{\mathsf{HTA}}(i, t)$.*

Proof: *It is easy to show that there exists a term $\bar{t}$ where $\omega_{q,\bar{t}} \cdot \omega_{i,\bar{t}} \geq \omega_{q,t} \cdot \omega_{i,t}$ for all terms t. Then, by definition of λ, we can show that condition (3.10) holds:*

$$\mathcal{S}_{qu}(q, i) = \sum_{t \in V} \omega_{q,i} \cdot \omega_{i,t} \leq \lambda \cdot \omega_{q,\bar{t}} \cdot \omega_{i,\bar{t}}$$

Maximum Query Weight (MQW) Since item weights are normalized, we can show that there exists at least one term t where the following condition is true:

$$\mathcal{S}_{qu}(q, i) \leq \omega_{q,t} \tag{3.11}$$

This leads to the corresponding upper bound condition and candidate set $C_{\mathsf{MQW}}(i,t)$:

$$\mathsf{MQW} : \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot \omega_{q,t} > \mathcal{S}_{min}(q)$$

Theorem 4 (Correctness of **MQW**). *For all queries q, all items i and all ranking functions $\mathcal{S}_{tot}(q,i)$, if $q \in U(i)$ then there exists at least one term t shared by q and i, such that $q \in C_{\mathsf{MQW}}(i,t)$.*

Proof: *Let $\bar{t}$ be the term shared by q and i with the maximum query weight $\omega_{q,\bar{t}}$. Then, by the fact that all item weights are normalized, we can proof condition (3.11) :*

$$\mathcal{S}_{qu}(q,i) = \sum_{t \in V} \omega_{q,t} \cdot \omega_{i,t} \leq \omega_{q,\bar{t}} \cdot \sum_{t \in V} \omega_{i,t} = \omega_{q,\bar{t}}$$

Theorem 5 (global correctness). *For all items i and all queries $q \in U(i)$,*

1. *there exists at least one term t shared by q and i such that condition* $\mathsf{LUB} \wedge \mathsf{HTA}$ *is satisfied by q, and*
2. *there exists at least one term t' shared by q and i such that condition* $\mathsf{LUB} \wedge \mathsf{MQW}$ *is satisfied by q.*

Proof: *Conditions (1) and (2) directly follow from theorems 1, 3 and 4 respectively.*

Terms t and t' are not necessarily identical. Equivalently, the *intersection* of all candidate queries obtained by LUB and HTA over all terms contain all queries to be updated :

$$U(i) \subseteq \bigcup_{t \in i} (C_{\mathsf{LUB}}(i,t) \cap C_{\mathsf{HTA}}(i,t))$$
$$U(i) \subseteq \bigcup_{t \in i} (C_{\mathsf{LUB}}(i,t) \cap C_{\mathsf{MQW}}(i,t))$$

Finally, we also can show by a simple counter-example that it is not possible to combine all three conditions without "loosing" queries to be updated.

Figure 3.2 shows two additional constraints corresponding to the light blue and light green areas. The left blue rectangle contains all queries q where $\mathcal{S}_{min}(q) < \alpha \cdot \mathcal{S}_{ite}(i)$

for item i. The green area adds all queries q where $\mathcal{S}_{min}(q) < \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot \omega_{q,t} \cdot \omega_{i,t}$. Both conditions are sufficient but not necessary for a query to be updated.

3.2.2 Filtering with decay

The query indexing techniques and upper bound conditions described in Subsection 3.2.1 do not take account of the decay function. Recall that we only consider order-preserving decay functions where the decayed score $\mathcal{S}(i, q, \tau)$ of all items i in the top-k result $R(q, \tau, k)$ continuously drops for all queries q, promoting newly arriving items over older ones. A direct interpretation of decay in our query representation consists in continuously moving the points in each index $P(t)$ towards lower values (towards the left) parallel to the minimum score axis (x-axis). Recall that decay does not change the item order which allows us to apply the decay function directly on the minimum score value of each query without considering its top-k result.

To avoid continuously updating $P(t)$ we apply the backwards decay technique proposed in [CSSX09]. This solution computes, stores and compares all scores with respect to some fixed reference time instant τ_0 used as a landmark. In particular, all query indexes $P(t)$ are maintained with respect to a constant time instant τ_0 and all queries in these indexes are only updated when their minimal scores change. The basic idea to achieve this is the following: Suppose that a new item i is published at time instant τ_i. The total score of a query q at τ_i is $\mathcal{S}_{tot}(i, q)$. In order to use this score with the corresponding query indexes, we have to calculate the inverse decayed score $decay^{-1}(\mathcal{S}_{tot}(i, q), \tau_i - \tau_0))$, which corresponds to the hypothetical score value that should have been assigned to the query-item pair at time instant τ_0:

$$decay(decay^{-1}(\mathcal{S}_{tot}(i, q), \tau_i - \tau_0), \tau_i - \tau_0)) = \mathcal{S}_{tot}(i, q)$$

Observe that the inverse decay function, always increases score values. This also implies that the minimum scores of items will always increase in time, since these, too, are computed with respect to the landmark τ_0. Projecting all scores and constraints on a given time instant τ_0 allows us to immediately compare and decide whether an arriving item updates or not a given query. We will see in Section 3.3 the practical impact of this solution on the underlying data structures.

3.3 Indexes

In this section we propose a number of in-memory index structures based on the two-dimensional representation and the linear constraints presented in Section 3.2. All indexes are designed by taking into account particular data characteristics and a common processing scheme. First, in order to be able to manage large term vocabularies, we follow a traditional inverted file approach mapping each term to a corresponding *inverted grid* of queries (we use the term *grid* opposed to the term list in the one-dimensional case). Second, each query q is encoded in the inverted grid of each terms $t \in q$ by a couple $(\mathcal{S}_{min}(q), \omega_{q,t})$. Since $\omega_{q,t}$ is constant, queries (data points) only move horizontally on the $\mathcal{S}_{min}(q)$-dimension. This simplifies the update process and favors the decomposition of the two-dimensional space into a list of horizontal *grid lines* for optimizing update cost. Observe, also, that each query minimal score update has to be done in all inverted grids of all remaining query terms. This obviously increases the importance of using data structures optimizing updates. Finally, as argued in Section 3.2.2, decay is computed with respect to a fixed landmark and does not cause any updates. However, this also leads to a monotonic unbounded increase of the indexing space – the minimal score of a query monotonically increases in time – which leads to the need for more dynamic data structures and incremental memory allocation. We will discuss this problem separately for each implementation.

Following the previous observations, the query filtering process can be summarized as follows. On item arrival, the corresponding inverted grids are retrieved for all item terms through the inverted file (hashtable). Then, for each grid, we start scanning its grid lines from the left to the right until the corresponding linear upper bound constraints presented in Section 3.2. For each candidate query q visited in this way, the system computes its total score and checks if its top-k list is updated by the new item. If this is the case, the minimal score of q increases (due to the insertion of the new item) and q is moved to its new position of *all* inverted grids corresponding to the query terms. Observe that, for the same item, a query can be visited several times, but it cannot be moved twice.

In the following we will describe four solutions for implementing inverted grids. The differences between these solutions lie in the choice of the data structures for

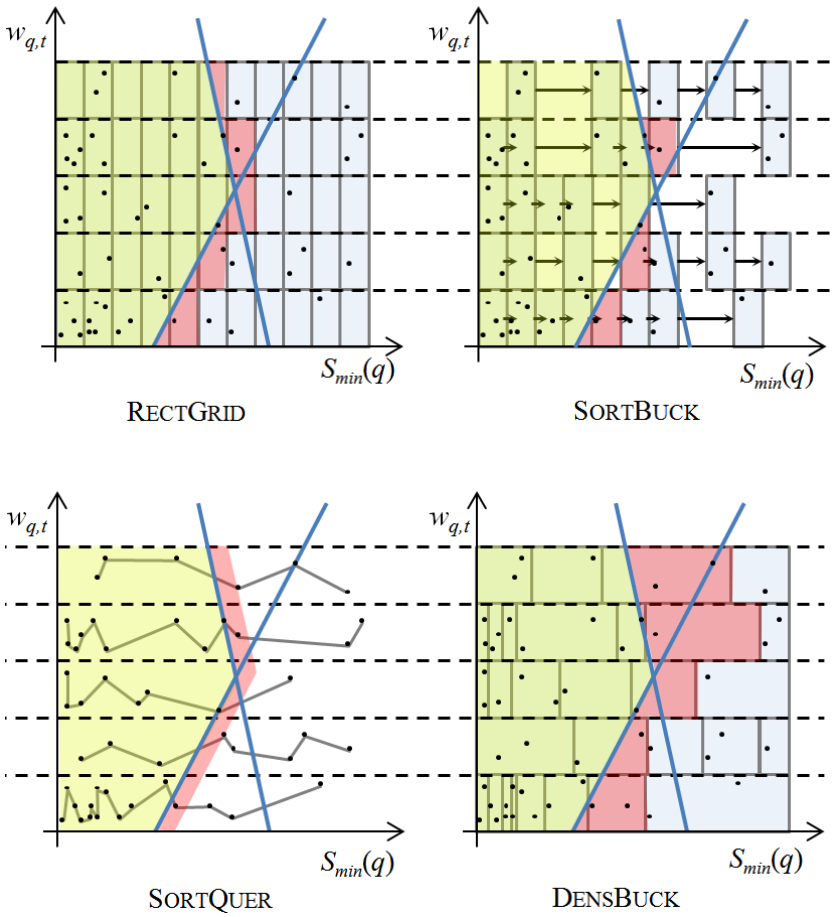


Figure 3.3: Query indexes

implementing grid lines. They can be compared following the standard evaluation criteria for data indexes: evaluation cost (including scanning, filtering), memory size and index update cost. All data structures are illustrated in Figure 3.3 and described in more detail below. Observe that all sub-figures contain the same queries and have been split into the same number of grid lines of constant height.

3.3.1 Rectangular Grid

RECTGRID is a two-dimensional array of equally sized cells as shown in Figure 3.3. Each cell is implemented as an unsorted set of queries called a *bucket*. For each item term

we can directly identify the array positions of all cells (buckets) that intersect with the constraint lines and which contain the right-most candidate queries. Moving updated queries is efficient in RectGrid since the bucket corresponding to the new position of an updated query can be calculated in constant time. Observe also that the change of the minimal scores might not necessarily change the position of a query in the grid. Concerning decay, since all scores are computed with respect to a fixed landmark time instant, the minimum (backward) scores of updated queries monotonically increase in time. Since this score increase is unbounded, it is not possible to define a static maximal array size. A solution consists in periodically recomputing *all* scores with respect to a new time instant, which might become very costly (lazy evaluation solutions might decrease this cost but are also more complex to be implemented). A better solution is to introduce more dynamic data structures as shown below.

3.3.2 Sorted Buckets

SortBuck introduces dynamic memory allocation by implementing grid lines as sorted lists of fixed sized buckets which contain at least one query. Buckets are ordered by their position in the grid (Figure 3.3). This data structure is obviously less efficient for updates since it is not possible to identify the bucket corresponding to a particular cell in constant time. In order to increase efficiency, the sorted lists are implemented as a Red-Black tree structure (the variation of B+ Trees optimized for main memory usage) obtaining a logarithmic update cost.

Notice that not all queries in the intersecting buckets of RectGrid and SortBuck are candidate queries and must be filtered individually. The cost influence of this precision loss (marked in red in figure 3.3) over the global matching cost might become important under certain distributions of term weights and minimal scores. Whereas decreasing the grid width increases this matching precision, it also increases the number of index updates.

3.3.3 Sorted Queries

In order to reduce matching precision loss, SortQuer directly maintains a list of queries sorted by their minimum scores (Figure 3.3). Query filtering is straightforward.

Updates are done in a similar way as for SortBuck and can be achieved in logarithmic time over the number of queries (using a tree structured ordered list implementation). An advantage of SortQuer is that a query's position only needs to be updated if its score gets higher than the minimal score of the next query. Observe, also, that this solution avoids the garbage collection cost generated by empty buckets.

3.3.4 Density Buckets

In RectGrid and SortBuck, buckets are of fixed width. Since we cannot make any assumption on the distribution of minimum score values (distribution depends on decay, term frequency, etc.), we obtain inhomogeneous collections of dense and sparse buckets. As discussed before, such biased distributions might have an important effect on matching preciseness. DensBuck adjusts bucket widths dynamically in order to maintain a minimum and maximum constant number of queries per bucket. Initially, all lines contain a single bucket of infinite width. After some time, these buckets will be split into sorted lists of buckets partitioning the whole inverted index space. This list is maintained dynamically similar to the nodes in a B-Tree structure (split and merge). Updates are performed in logarithmic time over the number of buckets. This solution better distributes queries within the buckets and it is equivalent to the previous solutions in terms of indexing and search complexity. Decreasing bucket density reduces search cost by improving precision but also leads to higher query update cost, as smaller buckets require more frequent split and merge operations. On the other hand, increasing density leads to the opposite effect (higher search cost against lower update cost). Our experiments in Section 3.6 confirm this trade-off between search cost and update cost and a low overall performance of the DensBuck index compared to the other index structures.

3.4 Approximate solution: a probabilistic model

The aforementioned representation and linear constraints determine, on item arrival, the candidate queries set and allow accurate retrieval of all updated queries. As an extension of our work in [VAC12], we have further analyzed the properties of the two-

dimensional representation and estimated the probability that an arriving item will update some candidate query given its position in a query index. Such knowledge can be exploited in order to improve performance by reducing the accuracy and provide an approximate solution to the query update problem where candidate queries with low *update probability* will not be examined. The value of the error introduced can be computed through the estimated update probability.

Let Q be the set of all queries stored in the system and $U(i)$ be the set of queries that item i will update.

Problem 2 (Update Probability Estimation)**.** *Given an item i, a term $t_0 \in i$ and the values $\mathcal{S}_{min}(q)$ and ω_{q,t_0} of any query q not necessarily in Q, find the probability* $\Pr(q \in U(i))$.

In other words, given any position in the query index of t_0, we want to compute the probability that any query q in that position will be updated by i.

On item arrival, when considering the query index of a term t of an item i, we have knowledge on the item score $\mathcal{S}_{ite}(i)$, as well as on the term weight and minimum score of any query in the index, given its position $(\mathcal{S}_{min}(q), \omega_{q,t})$. Given only this information we can infer that the update condition defined in Equation 3.3 is always true for the case where:

$$\alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot \omega_{q,t} \cdot \omega_{i,t} > \mathcal{S}_{min}(q)$$

This condition defines the light blue and green areas in Figure 3.2 where the update probability is 1. Furthermore, condition LUB defines an area where no queries will be updated (dark gray area in Figure 3.2). Consequently, we can immediately conclude that the update probability is 0 for that area.

All positions in the query indexes not included in these two areas (forming a triangle) have a non-zero and lower than 1 probability that they will be updated.

From the update condition we can directly conclude that:

$$q \in U(i) \Leftrightarrow \sum_{\forall t \in q \cap i - \{t_0\}} \omega_{q,t} \cdot \omega_{i,t} > c \tag{3.12}$$

where c can be computed in constant time given the $\mathcal{S}_{ite}$, ω_{i,t_0} and the position of q in the query index of t_0. Representing the unknown weights $\omega_{q,t}$ as random variables,

with $y_j = \omega_{q,t_j}$, the estimation of the probability can be formulated as:

$$E[P(q \in U(i))] = \int_0^1 ... \int_0^1 f_Y(y_1, ..., y_{|q \cap i|-1}) P(y_1, ..., y_{|q \cap i|-1}) \, \mathrm{d}y_1 ... \mathrm{d}y_{|q \cap i|-1}$$

where $f_Y(y_1, ..., y_{|q \cap i|-1})$ is the joint probability density function (PDF) of the random vector Y and $P(y_1, ..., y_{|q \cap i|-1})$.

Despite the numerous studies that have been published on the estimation of the weighted sum of random variables, the problem has not yet been solved for its general case. In [DK68] and in a more recent work in [SH09] the authors give a mathematical proof for the calculation of the value of the above formula making however, among others, the assumption that the random variables are uniformly distributed in [0, 1], an assumption which is not valid in this work. In our problem definition, we have not made any assumptions on the way term weights are assigned and thus, no assumptions on their distribution. Moreover, even if we did force a term weighting function with some given distribution, it can easily be proven that it is impossible to have a uniform distribution of normalized weights with query length different than 2. Although there are also a few works on the weighted sum of non-uniformly distributed random variables (Arcsin and Cauchy distribution [VA87], power distribution [Hom12]) they are all focusing on the case where there are exactly 2 variables y_1 and y_2.

Given the difficulty of solving this mathematical problem, we try instead to find the result using a sampling methodology, based on the Monte Carlo approach [JR83]. In particular, for every position in the query index we compute the constant c used in equation 3.12. Then, for a given number of repetitions we do the following: we replace in equation 3.12 each of the $\omega_{i,t}$ variables with the actual term weights of the given item and a random number in [0, 1] for the $\omega_{q,t}$ *based on the actual term weights distribution on the stored query workload*. Finally, we compute the percentage of these repetitions where the inequality of equation 3.12 was found to be valid. This percentage actually represents the probability of an error in case we do not check a query in that position. The result of probabilities in the grid found after this experiment for some given item is shown in Figure 3.4

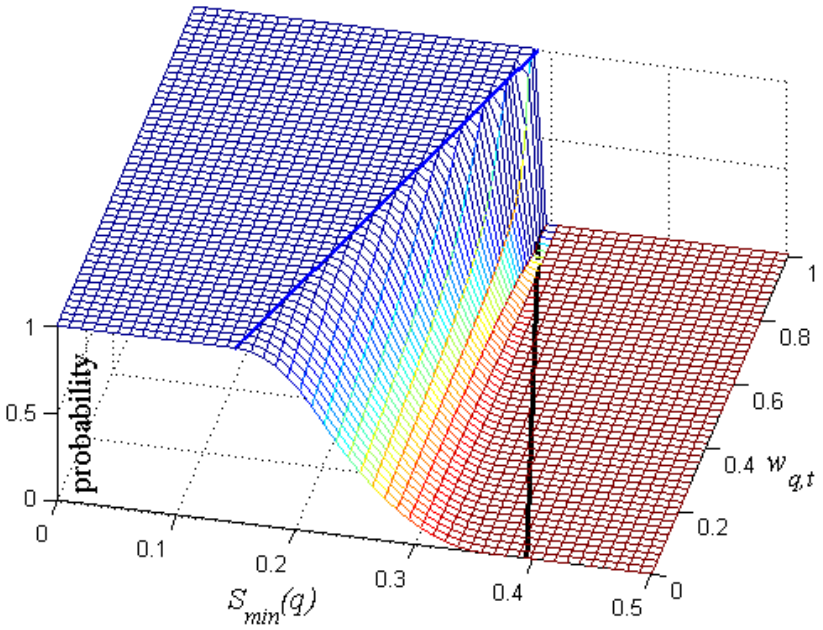


Figure 3.4: The update probability of queries found in some position ($\mathcal{S}_{min}(q)$, ω_{q,t_0})

3.5 Related work

3.5.1 Continuous top-k textual query processing

The algorithms most closely related to this work are the Incremental Threshold [MP11] and COL-Filter [HMA10]. A more thorough description of both these works can be found in Subsection 2.3.2.

Both Incremental Threshold and COL-Filter rely on monotonic and homogeneous ranking functions and use sliding window semantics without decay. Incremental Threshold is essentially a variation of the Threshold Algorithm (TA) [FLN03] and uses two inverted lists: (a) the first for coherently indexing the N most recently published items (sliding window) and (b) the second for indexing all registered top-k queries. This algorithm has been proven to be quite expensive in maintaining a valid view on the sliding window because of frequent index updates. As experimentally demonstrated in [HMA10], the Incremental Threshold's cost of retrieving candidate queries is most of the times worse than a naïve solution, where all queries containing the item terms are

scanned without any stopping condition. To overcome this limitation, COL-Filter defines a single inverted list of queries ordered by an appropriate one-dimensional ranking criteria that guarantees an effective early stopping condition.

Item importance has only been considered recently for query filtering [HMA12]. Item importance is estimated by applying a particular weighting scheme taking into account the frequency of terms in the queries. The total score of an item with respect to a given query then depends on its importance and its similarity. Whereas this reminds our definition of non-homogeneous total score, the way of how item importance is defined in [HMA12] makes the total scoring function again homogeneous for applying COL-Filter. Our work can accommodate the need for content-independent item scores resulting in non-homogeneous ranking functions, while its scalability gains even for the homogeneous case as presented in Section 3.6.

3.5.2 Multi-dimensional indexing

As we have seen in Section 3.3, retrieving candidate queries through our constraints is a spatial filtering problem in a two-dimensional space. Any incoming item defines a polygon for which all contained points of candidate queries need to be retrieved. Since, query updates affect the minimum score per result list they essentially redefine the position of a candidate query point by moving it towards to the right while keeping the same vertical position. Spatial indexes, such as Rectangular Grids, R-Trees, QuadTrees and k-d Trees, have be used in a similarly dynamic setting for retrieving moving objects continuously contained in a target area [KPH04]. However, these works rely on the strong assumption that the target search areas are known a priory. This is not the case of our spatial filtering where the polygons of interest are also continuously redefined based on the incoming items. Other spatial indexes proposed for moving objects either assume fixed velocity of the points [SJLL00], or compromise on the system's accuracy [CKP04]. Unlike these works in our setting (a) points frequently change positions and (b) points move only in a particular direction. For these reasons we have designed four indexes that take into account the peculiarities of points move and vertically partition in advance our two-dimensional space for all indexes.

3.5.3 Query dependent and independent scores in IR

Last but not least several pruning techniques for the inverse problem of top-k snapshot queries have been proposed in [LS03] using a ranking function for documents equivalent to ours. As queries arrive, the top-k documents are retrieved using knowledge on the weights of their terms as well as their PageRank [MRS08] score which is the equivalent of item importance in our work. In particular, reliable pruning technique proposed in [LS03] splits the collection of documents into two groups based on whether their term weight is higher than a threshold. Each of these groups is then sorted by the PageRank score. Results for arriving queries are then retrieved using a condition similar to LUB. Adapting this technique to our setting consists in split the queries into two groups depending on the query term weights and then sort them by the queries' minimum score. The SortQuer index can be seen as generalization of such a solution for more than two groups of queries.

3.6 Experiments

In this section we present an experimental evaluation over the indexes proposed in Section 3.3. We evaluate their performance under three different settings. We will start with the most simple scenario which considers only query dependent scores without decay (*homogeneous score without decay*). We will show the trade-off between matching time and memory cost over different index tuning parameters and compare our solution to the current state of the art solution (COL-Filter). We will then explore the effect of introducing decay on the matching performance (*homogeneous score with decay*). Our experiments terminate with the most general setting with decayed scores combining query similarity and query independent item scores (*non-homogeneous score with decay*).

3.6.1 Experiments setup

In order to run our experiments we use a real-world data collection and generate a corresponding set of queries. To the best of our knowledge there is no test-bed publicly available, including a stream of time-stamped items and user defined queries over the

same vocabulary and period. We ran our experiments over a dataset of 13,000 RSS items extracted from the RoSeS testbed [HVT+11] which contains items from 8,155 RSS feeds, including press, blog, forum feeds etc., collected from March 2010 to October 2010. Before running experiments we applied standard stemming, stop-word removal and HTML tag removal on the item contents. Observe that the size of the dataset (number of items) is not a scaling parameter in our setting which assumes a stream processing scheme where each item is processed in one pass and discarded afterwards. We then generated a synthetic query workload based on the vocabulary of our dataset. Using set of terms semantics for the items, we generated the queries by computing term co-occurrence and finding the *most frequent* combinations of 2 to 4 terms appearing in the dataset, aiming at evaluating index performance over a workload with many query relevant items. The final query workload was created by uniformly selecting the most frequent combinations of each query size.

For both, queries and items, we used the standard tf-idf weighting scheme with normalized weights. All experiments started by a warm-up matching period of 3,000 items in order to initialize the top-k results and minimum scores. The time measurements represent the average matching time required per item, over the remaining 10,000 items. The average matching time takes account of the time necessary for filtering all candidate queries (query filtering), updating the top-k results (query update) and updating the index according to the change of the minimal scores (index update). The k parameter, determining the size of the top-k results was set to 1. Higher values increase the number of query updates by decreasing the $\mathcal{S}_{min}(q)$ without any other particular effect on the index.

All algorithms were implemented in Java 6. Experiments were carried out on an Intel Core 2 Quad Q6600 @2.4 GHz, with 32-bit Windows 7 operating system, using 1GB of Java heap space.

3.6.2 Homogeneous score without decay scenario

In this first set of experiments we suppose a static environment (no decay), considering only query dependent scoring functions (α is set to 0). This simplification makes our problem statement compatible with existing continuous top-k query processing scenar-

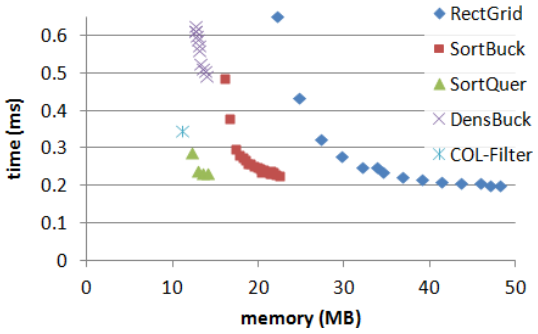


Figure 3.5: Trade-off between matching time and memory cost

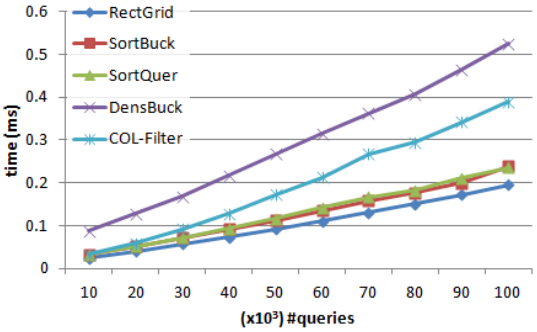


Figure 3.6: Matching time requirements over increasing number of indexed queries

ios. In particular, under this scenario we can compare our work to COL-Filter [HMA10], which is the state of the art solution and achieves better performance than the Incremental Threshold algorithm [MP09, MP11].

Besides comparing our solution with COL-Filter, we tune the four indexes proposed in Section 3.3 and evaluate the influence of their configuration parameters on matching time performance and memory cost. These configurations parameters are: (i) the number of lines (horizontal partitioning) for all four indexes (ii) the number of columns (vertical partitioning) for RectGrid and SortBuck and (iii) the bucket density for DensBuck.

As mentioned in section 3.3, increasing the degree of partitioning decreases the candidate filtering error (red zone in Figure 3.3), but at the same time this also increases update cost (updated queries move more frequently between buckets) and memory requirements (empty buckets, data structure overhead). In our first experiment (Figure 3.5), we capture this trade-off between matching time and memory cost. First we must notice that for all four indexes increasing the number of divisions (lines/columns) naturally led to better matching time requirements, but after some point this performance was worsening. This is explained by the fact that after a certain partitioning threshold, the performance gain from the higher candidate filtering precision cannot absorb the performance loss due to empty buckets and/or internal data management overload. Figure 3.5 therefore, represents for each index the skyline of the optimal matching time and memory requirements pairs, after varying the tuning parameters. Differently

formulated, each point (x, y) reflects the best average per item matching time y the index can achieve using not more than x MB of memory. Finally, note that COL-Filter has no particular configuration parameters which might influence memory usage and is therefore represented by a single point.

RectGrid requires more memory than all other indexes to achieve the same performance and COL-Filter uses less memory than all other indexes however, with non-optimal time performance (single cross on the left). The additional space required by RectGrid can be explained by the static memory overhead of the underlying array structure which is which independent to the actual number of indexed queries and non-empty buckets. SortBuck has a more clever dynamic usage of space, depending only on the buckets created. The goal of DensBuck was to obtain a better control over the number of queries per bucket (bucket density) which improves memory cost. However, as shown in the figure, the time performance results are disappointing compared to the other solutions. In this index, changing bucket density introduces a trade-off between filtering accuracy (search cost) and the number of time consuming bucket split and merge operations (update cost). In both cases, the additional time requirements lead to poor overall performance.

Finally, we can see SortQuer index, which directly generates lists of queries achieves the best matching performance with low memory. For this same reason, COL-Filter has slightly less memory requirements than even SortQuer (10-15%). Recall that COL-Filter does not consider two dimensions, but only stores sorted lists to maintain the equivalent of a grid in our solutions. Finally, we can see that all our index structures (except DensBuck) can achieve an average matching time performance gain of 50% (with respect to COL-Filter) under different memory constraints.

For the rest of the experiments we have initialized all index parameters with the optimal values leading to the best matching time performance independently to their memory usage (as shown in figure 3.5, adding more memory has no performance benefit after a certain threshold). Table 3.1 summarizes these optimal values for each index.

Figure 3.6 shows the scaling performance of our four indexes, as well as that of COL-Filter over the number of queries stored in the system. We can observe that the average per item matching time of all indexes increases linearly with the number of indexed queries. Since each subset of queries is an unbiased sample with the same term

	lines	columns	density	memory (MB)
RectGrid	20	1800	-	47.14
SortBuck	20	4000	-	22.57
SortQuer	20	-	-	14.09
DensBuck	3	-	16	13.98
COL-Filter	-	-	-	11.22

Table 3.1: Tuning parameters used on the experiments

distribution as the original set, this curve also reflects the scaling behavior with respect to the candidate queries. We can see that the matching performance of SortBuck and SortQuer are more or less equivalent whereas SortBuck requires 60% more memory than SortQuer. Grouping queries in buckets reduces the number of position updates required after minimum score changes on the queries, but it also leads to a loss of preciseness in query candidate filtering. Finally, COL-Filter performance decreases with a higher linear rate and requires up to 125% more average matching time than RectGrid and up to 70% more than SortBuck and SortQuer. DensBuck clearly has the worst performance.

3.6.3 Homogeneous score with decay scenario

In the following set of experiments we evaluate the behavior of the indexes when applying linear and exponential decay. As explained in Section 3.3, all indexes are maintained with respect to a reference time instant τ_0 by using a backward decay function. This has as a side effect that the minimal score of a query is monotonically increasing in time. In our query index representation, this results in a time dependent unbound query point distribution on the minimal score.

Due to this dynamic change of the indexed space RectGrid, which is based on a statically bound array structure, cannot be combined with our backward decay solution. We also exclude from the tests COL-Filter, which is based on sliding windows for reflecting information freshness. As before, in this experiment, the value α is set to zero and only query dependent scores are considered.

Generally, faster decay means that the item scores (including the minimal score) in the top-k list of a query decrease more rapidly with respect to a new item and

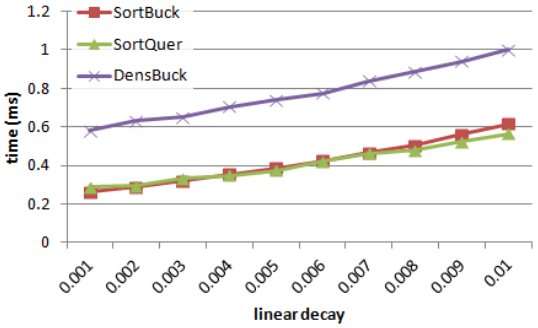


Figure 3.7: Time requirements over faster decay of scores(linear decay)

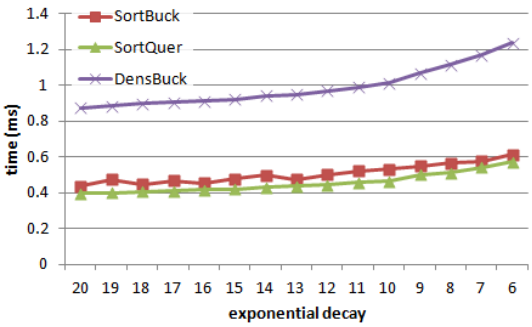


Figure 3.8: Time requirements over faster scores' decay (exponential decay)

arriving items are more likely to update relevant queries. Equivalently, when decay is applied to our constraints model, the three constraint lines continuously move toward the right and more queries have to be checked for updates. Figure 3.7 depicts the time requirements as scores decay faster, using a linear decay function. The x-axis in the diagram represents the continuous average score decrease per day. The chosen interval of values ($[0.001, 0.01]$) corresponds to 5 up to 20 updates on average per query per day. The behavior of the three indexes is linear on the decay rate / number of more updates on queries caused by the items. This indicates that even though more queries are scanned due to decay, the average cost in time per update over the whole set of queries remains more or less constant.

In Figure 3.8 we can see the equivalent diagram for exponential decay. The x-axis of the diagram represents the time required (in hours) in order to divide scores by 2 (half-life period). As before, values were chosen so as to have on average between 5 and 20 updates per query per day. For exponential decay, too we also observe that the behavior of the systems is proportional to the number of queries updated. Here, as well, no changes are observed in the relative position of the indexes performance, with DensBuck requiring twice the time of SortBuck and SortQuer having almost the same time performance as SortBuck.

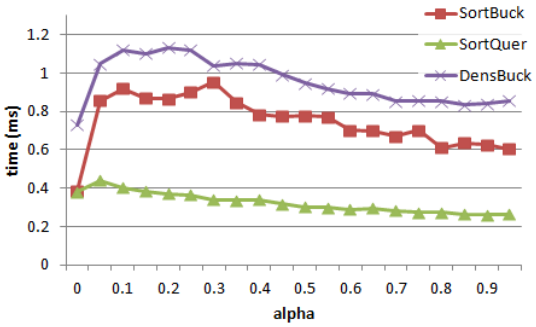


Figure 3.9: Time requirements over increasing α values, using linear decay

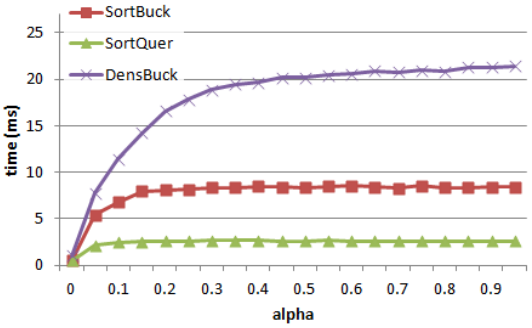


Figure 3.10: Time requirements over increasing α values, using exponential decay

3.6.4 Non-homogeneous scores with decay scenario

In the previous experiments we assumed homogeneous ranking function ($\alpha = 0$). In this final set of experiments we observe the behavior of the three index implementation over different values for α with linear and exponential decay. Item scores are generated as random values in $[0, 1]$ with uniform distribution. Decay was fixed to an average of 10 updates per query per day with homogeneous score $\alpha = 0$ (the final number of updates depends on α).

Figure 3.9 shows the performance of the three indexes over different values of α while applying linear decay. The time behavior shown, directly reflects the average number of candidate queries, but also of the ones updated per item. We can observe that for all indexes, switching from the homogeneous total score ($\alpha = 0$) to the non-homogeneous score ($\alpha > 0$) immediately increases the number of candidate queries per item. After this first peak, increasing α has the opposite effect, i.e. the number of candidate queries continuously decreases. The initial peak can be explained by properties of the scoring function employed. Generally there are many query-item pairs for which a low query-dependent score is assigned and which is not sufficient to lead to an update in the case of $\alpha = 0$. However, when item score is considered, a random number for the item score (in $[0, \alpha]$) is added to the total score, increasing the expected total scores and the probability of queries to be updated. The fact that after the first peak, the number of query updates decreases can be explained by the properties of the linear decay function: all scores (higher and lower) decrease by the same amount after a given time period,

forcing lower scores to go faster to a zero minimum score and thus, be updated by any relevant item (we have a higher number of updates when the average total score is low). From the way we have selected the item score, item scores are generally higher than similarity scores, so higher values of α (weight of item score) lead to higher values of total scores, and as explained, in a higher number of updates. Similar observations can be made on Figure 3.10 where exponential decay is applied.

3.6.5 Conclusions on the experiments

Our experiments show that depending on the amount of available memory, we can choose between SortQuer which achieves good performance with low memory and RectGrid for a 10-20% performance gain with a high memory cost (SortBuck is situated between these two indexes). Compared to COL-Filter (which only works for homogeneous ranking functions), SortQuer is up to 50% faster using only 15% more memory. We have also observed that partitioning the space in more homogeneously sized partition (in terms of number of queries), as it is done by DensBuck, does not improve per item, neither time performance nor memory cost. All three indexes supporting decay scale linearly with the number of queries. Matching time increases proportionally to the number of top-k list updates and is not affected neither by the decay rate nor by the linear combination of the query-independent and the query-dependent scores (α). Consequently, once the desired update rate for the users' queries has been fixed, the total scoring and the decay function, which affect immediately the results retrieved by the users, can be freely chosen without affecting the performance of the system. A final observation is that grouping of queries in buckets does not lead to a better performance. This observation is also confirmed by the experiments conducted in [HMA10] comparing COL-Filter and POL-Filter.

3.7 Summary

In this chapter we have introduced a new class of continuous top-k textual queries featuring dynamic *non-homogeneous* total ranking functions which combine information freshness, query dependent text similarity and item importance scores. Existing con-

tinuous top-k textual query processing systems [MP09, HMA10, MP11, HMA12] are based on variations of the Threshold Algorithm [Fag02] and a one-dimensional ordering of queries which can only be defined by using *monotonic* and *homogeneous* ranking functions. In this respect, the main contributions of this work are the following:

- Based on a new two-dimensional inverted query indexing scheme, we have explored efficient score bounds which drastically prune the search space of all candidate query top-k lists that are updated by the arrival of a new item. We have also proven the local optimality and soundness of these bounds.
- We have then introduced and compared different in-memory index structures implementing our spatial filtering conditions over dynamic query scores. Similarly to spatial indexes for moving objects [SJLL00, KPH04, CKP04], we have taken into account the particular moving behavior of query points to vertically partition in advance the two-dimensional search space in the design of our indexes.
- We have provided a thorough experimental evaluation of the memory/matching time trade-offs implied by these index implementations. An important result is that all four solutions exhibit a linear scaling behavior with respect to the number of queries matching an item, independently of the used ranking function. More importantly we have shown that all our proposed indexes scale linearly with respect to the number of updates (results). We have also shown that the index structures presented generalize state of the art solutions for the homogeneous case (COL-Filter), with our SortQuer index achieving 50% less time requirements while using only 15% more memory.

4 Real-Time Search Considering Feedback

In Chapter 3 we have focused on the online filtering of streaming information and have proposed a number of solutions for implementing the Item Handler component [VAC12], supposing that the query-independent item importance is assigned on the item's arrival and remains static. This assumption, however, is no longer valid if feedback signals on items, like ratings or shares, are additionally considered. Attempting to re-use our proposed solutions for the Item Handler to maintain query results over feedback signal streams would lead to a rather inefficient solution. It would, in fact, require the re-evaluation of each item i after every *event* (i.e. feedback signal) it receives, regardless of how small its impact will be. Even for a small number of additional updates, it would require the re-computation of all queries that should contain it, including the ones that already do. Consequently, applying this approach in a highly dynamic environment with massive amounts of feedback information, would lead to a poor overall performance.

In this chapter, we focus on the Event Handler component of our proposed architecture (Figure 4.1). More precisely, we consider the evaluation of continuous top-k queries over both *web information* and *real-time feedback signals*, based on generalized scoring functions considering text relevance, user interaction on published information and time decay. We will first show that current top-k query processing techniques presented in the literature are not designed for highly dynamic scores taking account of real-time web signals generated by user feedback. We will then propose a new generic architecture and a family of adaptive algorithms for efficiently processing continuous top-k queries with highly dynamic scores. We experimentally evaluate these algorithms over a real-world dataset of 23 million tweets and retweets collected during a 5-month period and compare the influence of dynamic scores in the performance of our algorithms on different workloads.

The rest of the chapter is organized as follows. Section 4.1 gives a formal definition of the problem. Section 4.2 presents our proposed solution and pruning techniques for event matching. Then, Section 4.3 describes a number of index implementations that apply these techniques. Section 4.4 provides the experimental evaluation. Finally, Section 4.5 summarizes the main contributions of this work.

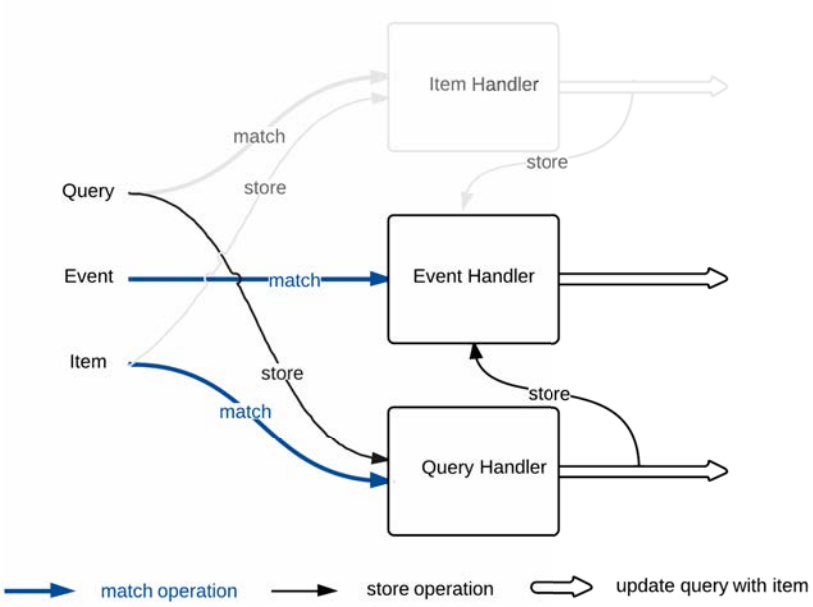


Figure 4.1: Focusing on the Event Handler component. Incoming events are handled in the Event Handler component which retrieves potential updates

4.1 Problem statement

In this section we formally define the notion of continuous real-time top-k queries with feedback signals. In fact, we extend our problem formulation for the Item Handler, presented in Section 3.1, to additionally consider *event streams* and *dynamic scoring functions*.

4.1.1 Query, item and event streams

The general data model builds on a set of items I (tweets), a set of search queries Q and a set of events E (feedback signals). Each event $e \in E$ concerns exactly one item i called the target of event e and denoted by $target(e)$. The set of all events concerning an item i is denoted by $\mathcal{E}(i)$.

To take account of information flow and real-time signals, we also assume a time-stamping function

$$ts : I \cup E \cup Q \rightarrow T$$

which annotates each item, event and query with their creation timestamp. This function formally transforms all sets I, E and Q into streams. For the sake of simplicity, we assume in the following that I, Q, E denote the sets of all items, queries and events that have been published until a fixed time instant τ.

4.1.2 Scoring functions

In this chapter, we extend the scoring function studied in Chapter 3, considering static query-item relevance ($\mathcal{S}_{qu}(q,i)$) and static query-independent item scores ($\mathcal{S}_{ite}(i)$), with a dynamic, query-independent *aggregated event score* which reflects the score changes to an item i triggered by the arrival of events $\mathcal{E}(i)$ that have i as target. We denote by $\mathcal{S}_{ev}(e)$ the score of a given event e for its target item i. This score can reflect for example, the importance of the type of event (e.g. if it is a comment, a share or a click), the importance of the user that issued it etc. Then we can define the *aggregated event score* $\mathcal{S}_{dyn}(i)$ of an item i as the sum of all event scores of events $e \in \mathcal{E}(i)$ with target i (known at the fixed instant τ):

$$\mathcal{S}_{dyn}(i) = \sum_{e \in \mathcal{E}(i)} \mathcal{S}_{ev}(e) \tag{4.1}$$

Like in the definitions of Chapter 3, we also assume that the query, item and event scores are positive and static. The dynamicity of the aggregated event score $\mathcal{S}_{dyn}(i)$, and therefore the total score, as well, derives for the arrival of new events, with i as target.

The a *total score* combines the query, item and aggregated event score into a dynamic function: $\mathcal{S}_{tot}(q,i,\tau)$ considering the event scores received up to time instant τ for item i:

$$\mathcal{S}_{tot}(q,i,\tau) = \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot \mathcal{S}_{qu}(q,i) + \gamma \cdot \mathcal{S}_{dyn}(i) \tag{4.2}$$

Observe that events affect the scores of items independently of the queries.

4.1.3 Score decay

To take account of the freshness of information, we assume the use of a decay function. Similarly to our definition for the Item Handler component (see Subsection 3.1.2), we consider the use of an order-preserving decay function over the query-item scoring function of Equation 4.2:

$$\mathcal{S}(q, i, \tau) = decay(\mathcal{S}_{tot}(q, i), \tau - \tau_i)$$

As discussed in Subsection 3.2.2, it is possible to avoid continuously updating the scores, changing due to time decay, by using backwards decay techniques: all scores are computed, stored and compared with respect to some fixed reference time instant τ_0 used as a landmark. This technique allows us to immediately compare all scores without performing any additional transformation.

4.1.4 Continuous top-k queries

Similarly to our definitions in Chapter 3, this total score $\mathcal{S}_{tot}(q, i, \tau)$ defines the result (semantics) of each query $q \in Q$. More precisely, the top-k result of some continuous query q (at a fixed time instant τ), denoted $R(q, \tau, k)$, is a subset of maximally k items $i \in I$ with a strictly positive query score $\mathcal{S}_{qu}(q, i) > 0$ and with a maximal global score $\mathcal{S}_{tot}(q, i, \tau)$, i.e. there exists no other item $i' \in I$ outside the result with a higher global score $\mathcal{S}_{tot}(q, i', \tau) > \mathcal{S}_{tot}(q, i, \tau)$.

We can now state the following general problem that has to be solved :

Problem 3. [CONTINUOUS REAL-TIME TOP-k QUERY EVALUATION] *Given a set of queries Q, a global score function $\mathcal{S}_{tot}$, an item stream I and an event stream E, maintain for each query $q \in Q$ its continuous top-k result $R(q, \tau, k)$ at any time instant τ.*

Let $\mathcal{S}_{min}(q, \tau)$ denote the (unique) *minimal score* of query q (at time instant τ): $\mathcal{S}_{min}(q, \tau) = min(\{\mathcal{S}_{tot}(q, i, \tau) | i \in R(q, \tau, k)\})$. Then, under certain constraints (order preserving decay function), the previous result maintenance problem can be decomposed into two (discrete) continuous item and event matching problems taking only account of the arrival of new items and events:

Problem 4. [CONTINUOUS TOP-k ITEM MATCHING] *Given a set of queries Q and a global score function $\mathcal{S}_{tot}$, identify for each new item i all queries $U(i)$ where $i \notin R(q,\tau,k)$ and $\mathcal{S}_{tot}(q,i,ts(i)) \geq \mathcal{S}_{min}(q,ts(i))$. We will call $U(i)$ the updates triggered by item i.*

Problem 5. [CONTINUOUS TOP-k EVENT MATCHING] *Given a set of queries Q and a total score function $\mathcal{S}_{tot}$ identify for each new event e all queries $U(e,\tau)$ respectively where $target(e) \notin R(q,\tau,k)$ and $\mathcal{S}_{tot}(q,target(e),ts(e)) \geq \mathcal{S}_{min}(q,ts(e))$. We will call $U(e,\tau)$ (or for simplicity $U(e)$) the updates triggered by event e .*

Problem 3 has already been extensively studied in literature in the field of Information Retrieval and Problem 4 has been addressed in Chapter 3. In the rest of this chapter we will mainly concentrate on Problem 5.

Example 1. *Figure 4.2 shows the evolution of the top-2 result of some q for item set $\mathcal{I} = \{a,b,c,d\}$. The score evolution of each item is represented by a step-wise monotonically increasing line where each increase corresponds to the arrival of some event (aggregated event score). The minimum score of q is represented by a bold line. Each item has an initial positive query score and the query result is updated seven times: at the arrival of a new item (item match at τ_a, τ_b and τ_d) as well as at the arrival of a new event (event match at τ_1, τ_2, τ_3 and τ_4). Observe that the arrival of a new items does not necessarily trigger a query update (e.g. arrival of c) and an item can disappear and reappear in the result (item b disappears at τ_d and reappears at τ_3).*

Since we consider only positive event scores, we can ignore events on items $i \in R(q,\tau,k)$ in the top-k result of q. Whereas these events can change the minimal score of q, this score can only increase and the top-k result does not change as long as there are no events on other items.

4.2 Event Handler algorithms

Figure 4.1 presents our target overall functionality consisting of a Query Handler component, answering queries as they arrive, an Item Handler, updating query results on

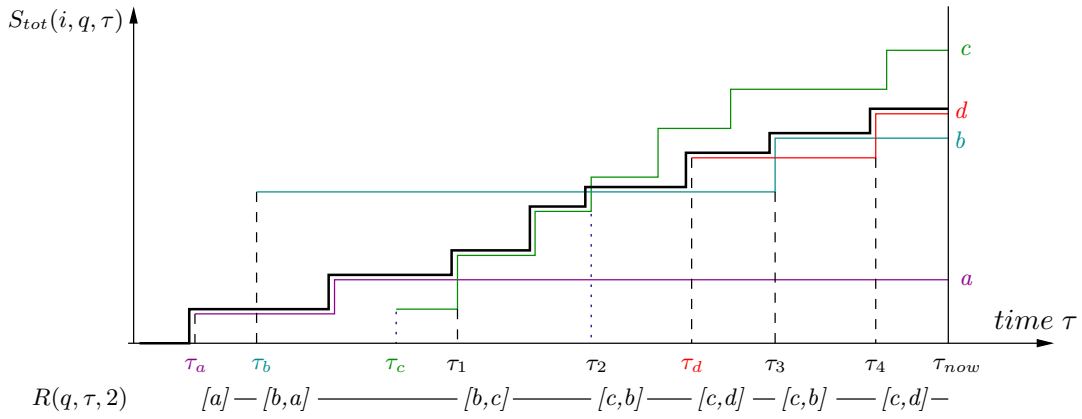


Figure 4.2: Top-2 result evolution of query q

item arrival, and an Event Handler, retrieving updates triggered by events. In Chapter 3 we have presented a general representation of continuous queries and a number of solutions for implementing the Item Handler component. In the following we focus on the Event Handler and describe two event matching algorithms which enable the efficient evaluation of feedback events on a real-time search system.

4.2.1 The *AR* algorithm

The first algorithm is called *All Refresh* (*AR*, see Algorithm 1). In this first naïve approach, we consider the evaluation of all incoming events in the Item Handler component: for each incoming event e, we increase the query-independent score of the corresponding item $target(e)$ by the value designated by e and re-evaluate the item as if it was a newly published one (see Figure 4.3). This evaluation will actually retrieve a superset of the desired result: it will additionally retrieve queries that already contain $target(e)$ on their top-k result set. On a last step of the event evaluation procedure of *AR*, we iterate through the results and remove such queries, thus obtaining the correct result set of updates.

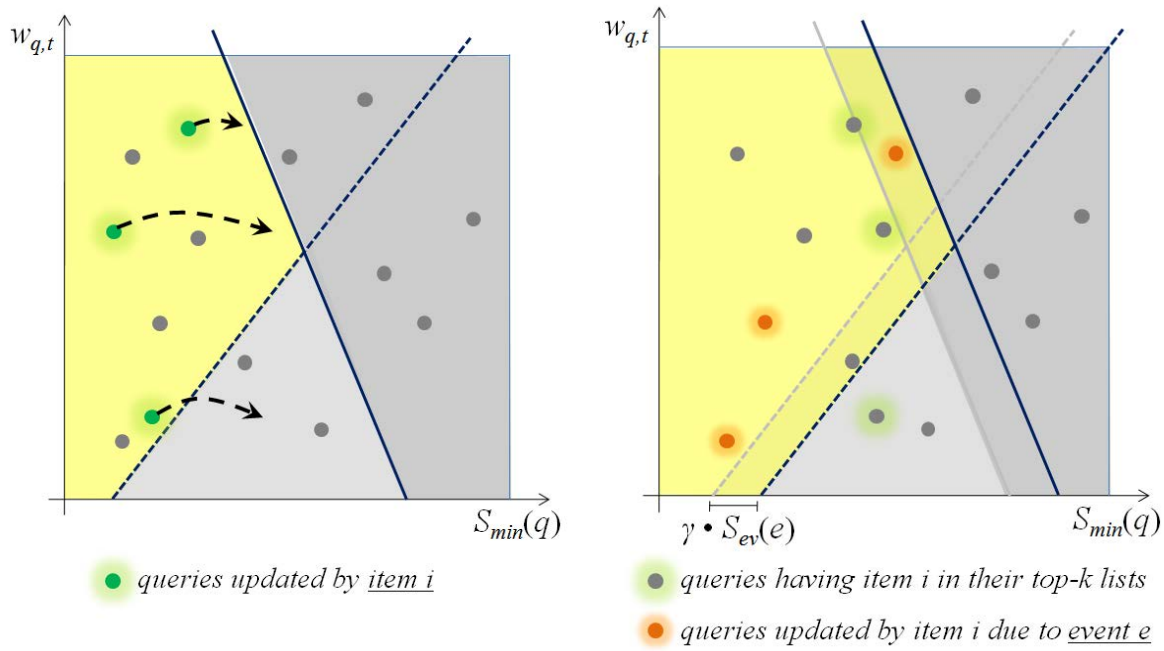
Function `EH.matchEventSimple` of algorithm *AR* computes the updates triggered by event arrivals by increasing the dynamic score of the corresponding item and re-evaluate it in the Item Handler (through function `IH.processItem`). Function `RTS.update` is responsible for updating a set of queries by a given item (a newly pub-

Algorithm 1: The AR algorithm

```
1  IH.processItem(i: Item)
2      Q(i) := ∅;
3      U(i) := IH.matchItem(i);
4      RTS.update(U(i), i);
5  EH.processEvent(e: Event)
6      i := target(e);
7      U(e) := EH.matchEventSimple(e);
8      RTS.update(U(e), i);
9  RTS.update(U(i): set(Query), i: Item)
10     for q ∈ U(i) do
11         i_k := R(q, τ, k)[k];
12         R(q, τ, k) := R(q, τ, k) + i − i_k;
13         Q(i) := Q(i) + q;
14         Q(i_k) := Q(i_k) − q;
15 IH.matchItem(i: Item)
16     as implemented in [VAC12]
17 EH.matchEventSimple(e: Event)
18     i := target(e);
19     S_dyn(i)+ = S_ev(e);
20     return IH.matchItem(i) − Q(i);
```

lished item or the target item of an event).

Looking into more detail Algorithm AR, each new item i is processed by procedure `IH.processItem` which initializes the set of matching queries $Q(i)$ (line 2). It then matches all queries to be updated calling function `IH.matchItem` (line 3) and finally calls function `RTS.update` to update these queries. We do not detail `IH.matchItem` here and suppose that we can use any continuous top-k algorithm, similar to the one presented in our work in [VAC12] (Chapter 3), which is able to evaluate the global scoring function $\mathcal{S}_{tot}$. Function `RTS.update` identifies and removes for each query q to be updated, the last item i_k in the query result (lines 11 and 12), adds the new item (line 12) and updates the matching query sets $Q(i)$ and $Q(i_k)$ of i and i_k (lines 13 and 14).



(a) Evaluation of item i at its publication time τ_i

(b) Evaluation of event e with $target(e) = i$ at its publication time τ_e on the Item Handler

Figure 4.3: (a) The arrival of an item i triggers the update of three queries (in green) which are moved into their new positions. (b) Supposing the evaluation of an incoming event e for item i ($target(e) = i$) in the Item Handler, three new updates are retrieved (in orange). Observe that the new constraint lines for the item re-evaluation are the ones of the first evaluation of i at τ_i, moved on the right by a distance of $\gamma \cdot \mathcal{S}_{ev}(e)$. Updates caused by the arrival of an event e can be in *any position of the grid designated in yellow.*

Theorem 6. *Algorithm AR is correct for positive event scores.*

Proof. We have to show that both functions, `IH.processItem` and `EH.processEvent`, guarantee that all items i and all queries $q \in Q$, if $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{min}(q)$, then $q \in Q(i)$ (condition *AlgCorr*).

We first show that for a given item i there exists no query $q \in Q - Q(i)$ where $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{min}(q)$ after the execution of (1) `IH.processItem`(i) and (2) `EH.processEvent`(e) processing an event e with target item i. For (1) `IH.processItem`(i), by definition, the item matching function `IH.matchItem` returns

all queries where $\mathcal{S}_{tot}(q,i) > \mathcal{S}_{min}(q)$ and after processing `RTS.update` (i), $Q(i)$ contains all these queries these. The same argument holds for `EH.processEvent`(e) which also calls `IH.matchItem` (through function `EH.matchEventSimple` and then `RTS.update` (i) after the update of the dynamic score. We can then show that condition $AlgCorr$ holds for item i until to its next event. This is easy to show by the fact that $\mathcal{S}_{tot}$ is monotonically increasing (positive event scores). First, either q is never removed from $Q(i)$ (trivial). Second, if q is removed from $Q(i)$ (line 14), we know that $\mathcal{S}_{tot}(q,i) < \mathcal{S}_{min}(q)$. Since $\mathcal{S}_{min}(q)$ is monotonically increasing and $\mathcal{S}_{tot}(q,i)$ only can be updated by an event with target i, the last condition holds until to the next event. □

Although the AR algorithm might have an acceptable performance on a system where events are quite rare, compared to the item's arrival rate, it would be quite inefficient on a real-time system with much user feedback signals, such as Twitter. In such a dynamic setting, with events arriving at high rates, we would expect that a single event on an item, e.g. a single retweet or favorite, will have, on average, a small impact on the set of queries that would receive it. However, by re-evaluating the item through the Item Handler, we would have to re-compute a potentially long list of queries that have already received i.

A more efficient solution would be to directly compute the required result set (without the old updates) which is equivalent to the set $U(e) = \{q \in Q | \mathcal{S}_{min}(q) \in (s_0, s_1]\}$ where s_0 and s_1 correspond to the global scores before and after adding the event score. Unfortunately, such an operation is not supported and cannot directly be implemented in an efficient way on the Item Handler implementation of Chapter 3. Observe for instance Figure 4.3: assuming that it would be sufficient to check only the righter part of the yellow region (the part that was not visited at τ_i in i's first evaluation), is mistaken. Updates can be found in any part at the left side of the constraint lines: those were queries that caused false positives in the original item evaluation (at τ_i).

In the following we propose instead a new algorithm which is implemented in the Event Handler component for efficiently processing incoming events independently to the Item Handler's implementation.

4.2.2 The R^2TS Algorithm

The main task of the Event Handler is to perform the event matching operation, i.e. retrieve all updates triggered by an arriving event. The efficient implementation of this task is based on the following observations:

- each item i has limited number of relevant queries q (with $\mathcal{S}_{qu}(q,i) > 0$) which might be updated by future events;
- each event increases the score of a single item and triggers a limited number of updates;
- the maximum aggregated score of an item depends on the scoring function and the number of events it will receive.

Based on these observations, we propose the *Real Real-Time Search* (R^2TS) algorithm, which is based on the following idea for matching an incoming event. As we have seen before, it is not efficient to trigger the Item Handler for recomputing all updates $U(e)$. In R^2TS instead, on the initial evaluation of an item i on the Item Handler, we additionally retrieve a number of *query candidates* for i, i.e. queries for which the total query-item score is not sufficient for them to be updated by i, but could potentially receive it in the future, if an adequate number of events increase the aggregated item score of i. In other words, the query candidates of an item are, ideally, the queries with a higher probability of being updated by that item due to incoming events.

The definition of query candidates highly depends on the query-item scoring function and more precisely on the way the events affect the total score. Given our scoring function (Equation 4.2) we observe that each incoming event increases by a additional value to the item's score. This leads us to a straight-forward way of choosing the candidates, which is by setting a threshold θ_i for each item i, indicating our prediction on the dynamic score i will receive by future events.

Definition 7. *Given a positive value θ_i, the candidates $\mathcal{C}_i \subset Q$ of an item i is the set of queries: $\mathcal{C}_i = \{q \in Q | \mathcal{S}_{min}(q) \in (S(q,i), S(q,i) + \gamma \cdot \theta_i]\})$.*

Definition 8. *θ_i-condition: On event e arrival, we say that the θ_i-condition holds for the corresponding item i iff the dynamic score of the item does not exceed the defined value θ_i: $\mathcal{S}_{dyn}(i) + \mathcal{S}_{ev}(e) < \theta_i$.*

For example, in Figure 4.3(b) if candidate queries are computed for the item i on its arrival (at τ_e) and given that the assigned value θ_i is greater than the score $\mathcal{S}_{ev}(e)$, it is guaranteed that the three queries in orange, i.e. those updated due to event e, will be in the candidates set.

Thus, as long as the θ_i-condition holds for an item i, we know that the updates triggered by any event e for i are in the candidate set $\mathcal{C}_i$, stored in the Event Handler. If however, the θ_i-condition does not hold, the set of candidates is no longer valid and needs to be recalculated through the Item Handler. The re-evaluation is similar to the one described for the naïve solution.

The overall candidates approach of R^2TS is presented in Algorithm 2 can is summarized in the following. Each new item i is assigned with an initial threshold θ_i (line 5) and refresh counter $r(i)$ (line 6). Threshold θ_i represents the maximal score change in the item's dynamic score due to future events between two refresh operations. Item i_{θ_i} is a copy placeholder of item i (line 28) with a virtual aggregated event score corresponding to the future event score of i before the next refresh. i_{θ_i} is then used to find all candidate queries, additionally to the updates, until the next refresh.

When a new event arrives, function `EH.matchEvent` computes the update set $U(e)$. It increases the target item's aggregate score and first checks if it necessary to refresh the candidates (lines 27 to 30). As long as threshold $\mathcal{S}_{dyn}(i) > r(i) \cdot \theta_i$, the result is obtained by the naïve solution and the candidate list is refreshed with the new aggregated threshold. When $\mathcal{S}_{dyn}(i) \leq r(i) \cdot \theta_i$, `EH.matchEvent` copies all candidate queries q where the minimum score of q is strictly smaller than the global score of i to the update set $U(e)$ (lines 31 to 34).

Theorem 7. *Algorithm R^2TS is correct for positive event scores.*

Proof. We have to show that both functions, `IH.processItem` and `EH.processEvent`, guarantee that for all items i and all queries $q \in Q$, if $\mathcal{S}_{tot}(q, i) > \mathcal{S}_{min}(q)$, then $q \in Q(i)$ (condition $AlgCorr$).

Algorithm 2: The R^2TS algorithm

```
1  IH.processItem(i: Event)
2      U(i) := IH.matchItem(i);
3      Q(i) := ∅;
4      RTS.update(U(i), i);
5      EH.initThreshold(θ_i);
6      r(i) := 0;
7  EH.processEvent(e: Event)
8      U(i) := EH.matchEvent(e);
9      RTS.update(U(i), i);
10 RTS.update(U(i): set(Query), i: Item)
11     for q ∈ U(i) do
12         i_k := R(q, τ, k)[k];
13         R(q, τ, k) := R(q, τ, k) + i − i_k;
14         Q(i) := Q(i) + q;
15         Q(i_k) := Q(i_k) − q;
16         C(i_k) := C(i_k) + q;
17         C(i) := C(i) − q;
18 EH.matchEvent(e: Event)
19     if θ_i = 0 then
20         i := target(e);
21         S_dyn(i)+ = S_ev(e);
22         return IH.matchItem(i) − Q(i);
23     else
24         i := target(e);
25         S_dyn(i)+ = S_ev(e);
26         if S_dyn(i) > r(i) · θ_i then
27             r(i)+ = S_dyn(i)/θ_i + 1;
28             i_θi := i;
29             S_dyn(i_θi) := r(i) · θ_i;
30             C(i) := IH.matchItem(i_θi) − Q(i);
31         U(e) := ∅;
32         for q ∈ C(i) do
33             if S_tot(q, i)>S_min(q, ts(i)) then
34                 U(e)+ = q;
35             return U(e);
```

We already have shown that the correctness condition holds for some item i and all queries q after the execution of function `IH.processItem` (proof of Theorem 6).

We first show that it also holds after the execution of function `EH.processEvent`. For $\theta_i = 0$ we can use the same argument as in Theorem 6 (if $\mathcal{S}_{tot}(q,i) > \mathcal{S}_{min}(q)$, then $q \in \mathcal{C}(i)$).

For a positive threshold $\theta_i > 0$, let

$$\mathcal{S}_{stat}(q,i) = \alpha \cdot \mathcal{S}_{ite}(i) + \beta \cdot \mathcal{S}_{qu}(q,i)$$

be the *static query-item score* of item i for query q. We know after executing lines 27 to 30 that

$$\mathcal{S}_{dyn}(i) \leq r(i) \cdot \theta_i \tag{4.3}$$

and the candidate set $\mathcal{C}(i)$ of i contains *all* queries $q \in Q - Q(i)$ where

$$\mathcal{S}_{min}(q) \leq \mathcal{S}_{stat}(q,i) + \gamma \cdot r(i) \cdot \theta_i \tag{4.4}$$

To finish, it is sufficient to prove that the obtained candidate set contains all queries q' which have to be updated: $U(e) \subseteq \mathcal{C}(i)$. By definition, for all queries q' to be updated holds

$$\mathcal{S}_{min}(q') < \mathcal{S}_{tot}(q',i) = \mathcal{S}_{stat}(q',i) + \gamma \cdot \mathcal{S}_{dyn}(i) \tag{4.5}$$

Then since $\mathcal{S}_{dyn}(i) \leq r(i) \cdot \theta_i$ and (Equation 4.5),

$$\mathcal{S}_{min}(q') \leq \mathcal{S}_{tot}(q',i) \leq \mathcal{S}_{stat}(q',i) + \gamma \cdot r(i) \cdot \theta_i \tag{4.6}$$

which means that $q' \in \mathcal{C}(i)$.

Then, similarly to the algorithm AR, we can show that condition 33 holds for item i until its next event. First, it is easy to see that q is never removed from $Q(i) \cup \mathcal{C}(i)$ (trivial from lines 17 to 15). Second, we know that $\mathcal{S}_{tot}(q,i) < \mathcal{S}_{min}(q)$. Since $\mathcal{S}_{min}(q)$ is monotonically increasing and $\mathcal{S}_{tot}(q,\, i)$ only can be updated by an event with target i, the last condition holds until to the next event. □

The challenge arising from this algorithm is twofold. Identify optimal threshold values θ_i for each item (see Subsection 4.2.3) and index the obtained candidate queries in a way that allows efficient retrieval of updates during the event matching operation (lines 31 to 34) (see Section 4.3).

4.2.3 Cost analysis

The choice of θ_i value controls the number of candidate refresh (item match) operations and has an important impact on the overall performance of the system. Before discussing the trade-offs between high and low values of θ_i, let us consider two extreme cases, where $\theta_i = 0$ and $\theta_i = +\infty$.

$\boldsymbol{\theta_i = 0}$ when threshold θ_i is set to zero, $\mathcal{C}(i)$ is empty and set $U(e)$ is computed by the Item Handler on each event arrival (naïve solution).

$\boldsymbol{\theta_i = +\infty}$ A positively infinite threshod $\theta_i = +\infty$ results in storing as candidates for i, the set of all relevant queries except $Q(i)$. This approach means that the candidates will never be refreshed by the Item Handler. However, this also results in a high overhead for storing, matching and updating query candidates in the Event Handler because of the potentially high number of false positives in the matching phase (lines 31 to 34). Observe also, even if there is no refresh phase, the candidate list of an item might be very often updated by procedure `RTS.update`.

From the previous observations, we can easily understand that higher values of θ_i minimizes the number of candidate re-evaluations through the Item Handler but also might generate candidate lists with a lot of false positive query candidates which will never receive the item as their update. On the other hand, low values of θ_i lead to more frequent costly candidate re-evaluations on the Item Handler. In our experiments in Section 4.4 we show that the total execution time for item and event evaluation on Twitter's stream can differ to up to an order of magnitude, based on the choice of θ_i. Both solution might be efficient if the number of events is low (low number of refresh and small candidate list respectively).

Ideally, a perfect estimation on the dynamic score the item will receive will enable the evaluation of the item only once through the Item Handler and then, all incoming

events could be matched in the Event Handler. However, this ideal solution might still not be optimal due to the higher average matching and maintenance cost for large candidate lists (this will also be illustrated by our experiments).

Formal cost model and optimization

In the following we show how the optimal value of θ_i for an item i can be determined based on an estimation on the final item's dynamic score and on the number of events the item will receive.

Let $cost(i, \theta_i)$ denote the total aggregated execution time for some threshold value θ_i. We denote by $N_i = |\mathcal{E}(i)|$ the estimated total number of events and by r_{θ_i} the number of necessary item match operations executed by function `EH.matchEvent`: $r_{\theta_i} = \lceil \theta_i^{max}/\theta_i \rceil$ for $\theta_i > \mathcal{S}_{ev}(e)$ and $r_{\theta_i} = N_i$ for $\theta_i = 0$. Observe that if $0 > \theta_i > \mathcal{S}_{ev}(e)$, we only know that $r_{\theta_i} < \lceil \theta_i^{max}/\theta_i \rceil$.

We will try find the optimal value for θ_i that minimizes this local cost function for each item. Under the assumption that items are independent, since each event corresponds to only one item, locally optimizing $cost(i, \theta_i)$ for every item i leads to a globally optimized evaluation cost.

The local cost value $cost(i, \theta_i)$ is the sum of the aggregated item matching cost (Item Handler), denoted $cost\text{IH}(i, \theta_i)$, the aggregated event matching cost (Event Handler), denoted $cost\text{EH}(i, \theta_i)$. More precisely,

- $cost\text{IH}(i, \theta_i)$: Each item is matched in the Item Handler index r_{θ_i} times: once for computing the initial update $Q(i)$ (independently of θ_i) and at each candidate list refresh, i.e. when the first event arrives and each time a new event arrives and θ_i events have been processed since the last refresh. The cost of every item matching operation depends on $costM(i)$ of finding the updates and the extra cost $costC(i, \theta_i)$ of retrieving and creating the candidate list in the Event Handler:

$$cost\text{IH}(i, \theta_i) = r_{\theta_i} \cdot (costM(i) + costC(i, \theta_i)) \tag{4.7}$$

- $cost\text{EH}(i, \theta_i)$: Each event is matched in the Event Handler as long as there is no refresh, i.e. $N_i - r_{\theta_i} + 1$ times. The cost of every call to the event matching

operation depends on the number of candidates $\mathcal{C}_{\theta_i}(i)$ and on the (constant) cost $costT$ of checking for an update in a single query-item pair. Supposing there is no early stopping condition when checking the candidates we obtain:

$$cost\text{EH}(i, \theta_i) = (N_i - r_{\theta_i} + 1) \cdot \mathcal{C}_{\theta_i}(i) \cdot costT \tag{4.8}$$

Observe that we do not take account of the cost for maintaining the candidate list. It is quite obvious that this cost also depends on the size of the candidate list (threshold θ_i) and we will show in Section 4.3 that the choice of the used index data structures might strongly influence this cost, in particular when the index is ordered for obtaining early stopping conditions during event matching.

Case 1: No candidates When $\theta_i = 0$, no candidates are maintained and all events are matched in the Item Handler. Since, for $\theta_i = 0$, $r_{\theta_i} = N_i$, $costC(i, \theta_i) = 0$ and $\mathcal{C}_{\theta_i}(i) = 0$ we obtain:

- $cost\text{IH}(i, \theta_i) = N_i \cdot costM(i)$
- $cost\text{EH}(i, \theta_i) = 0$

Case 2: One refresh When $\theta_i = \theta_i^{max}$, all event evaluations are performed on the Event Handler ($r_{\theta_i} = 1$):

- $cost\text{IH}(i, \theta_i^{max}) = costM(i) + costC(i, \theta_i)$
- $cost\text{EH}(i, \theta_i^{max}) = N_i \cdot \mathcal{C}_{\theta_i}(i) \cdot costT$

Finding the optimal θ_i

The value of $\mathcal{C}_{\theta_i}(i)$ depends on the distribution of query minimum scores. In the following we determine the optimal value of θ_i based on the assumption of a uniform query candidate distribution over all refresh iterations: $\mathcal{C}_{\theta_i}(i) = a \cdot \theta_i$, where a is a positive constant. We also assume that the candidate creation $costC(i, \theta_i)$ depends linearly on θ_i: $costC(i, \theta_i) = b \cdot \theta_i$. These assumptions simplify the calculation of the optimal θ_i, nevertheless, similar analysis can be performed assuming different distributions.

Given $r_{\theta_i} = \theta_i^{max}/\theta_i$ (we consider here the real value instead of the integer floor value), Equations (4.7) and (4.8) the total $cost(i, \theta_i)$ can be written as:

$$\begin{aligned} cost(i,\theta_i) &= \frac{1}{\theta_i} \cdot \overbrace{\theta_i^{max} \cdot costM(i)}^{c1} \\ &\quad + \theta_i \cdot \overbrace{(N_i+1) \cdot a \cdot costT}^{c2} \\ &\quad + \overbrace{\theta_i^{max} \cdot (b + a \cdot costT)}^{c3} \end{aligned}$$

We can minimize the *cost* function using the first derivative:

$$\frac{d(cost(i,\theta_i))}{d\theta_i} = -\frac{c_1}{(\theta_i)^2} + c_2$$

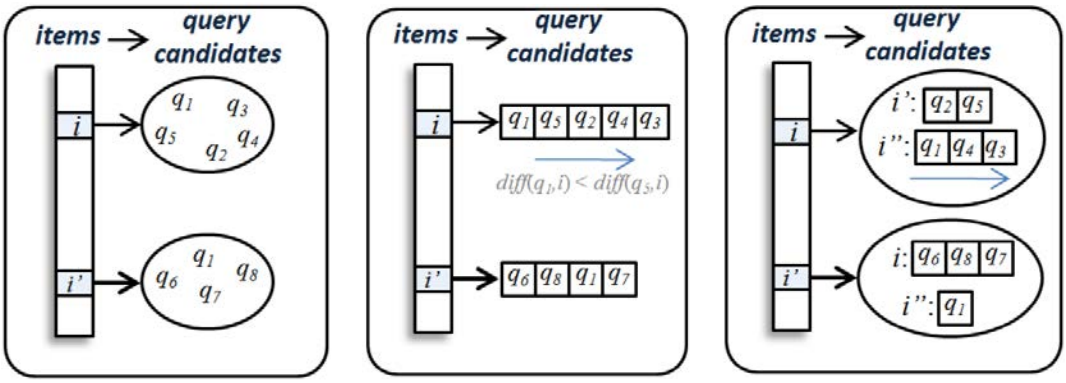
We can easily see that with $\theta_i \in (0, \theta_i^{max}]$ the derivative is negative when $\theta_i < \sqrt{c_1/c_2}$ and positive when $\theta_i > \sqrt{c_1/c_2}$. Consequently, function *cost* is monotonically decreasing in the interval $(0, \sqrt{c_1/c_2})$ and increasing in $(\sqrt{c_1/c_2}, \theta_i^{max}]$, thus making $\theta_i = \sqrt{c_1/c_2}$ the optimal value:

$$\theta_i^{opt} = \sqrt{\frac{\theta_i^{max} \cdot costM(i)}{(N_i+1) \cdot a \cdot costT}}$$

This equation essentially shows that θ_i^{opt} increases with the ratio $costM(i)/costT$ between the item matching cost and the event test cost.

4.3 Candidate indexing

The Event Handler stores query candidates (computed in the Item Handler) and matches incoming events against them, given that the θ_i-condition holds. In the following we refine the requirements for the operations of an Event Handler index and then, propose three categories of indexing schemes of the Event Handler and aim at optimizing the execution time of both storing and retrieval operations.



(a) (b) (c)

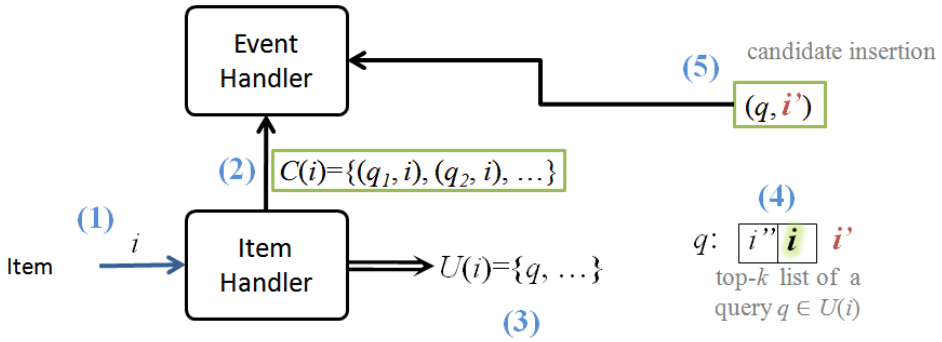
Figure 4.4: The (a) simple, (b) ordered and (c) item partitioning indexes.

4.3.1 Refining operation requirements

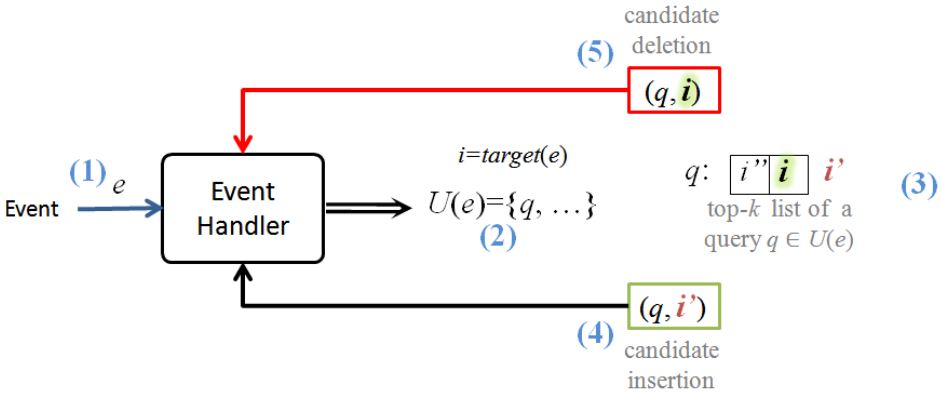
The main task of the Event Handler is to perform the event matching operation, i.e. retrieve updates triggered by any arriving event. Since each event e increases the score of a single item $target(e)$, the updates it triggers will only concern this item and a subset of its candidate queries. Thus, naturally, the building block of all following proposed indexes is a dictionary from each item to its set of candidates. The proposed indexes aim at organizing the *posting lists*, i.e. the candidates sets in a way that decreases the number of false positives encountered during event matching.

Besides the matching operation, it is also important for the index to efficiently support insertions and deletions. Candidate insertions can occur as the result of either the item evaluation or a query update detected in either the Item Handler or the Event Handler(Figure 4.5). In the first case, the Item Handler detects the query candidates for an arriving (or re-evaluated) item i which lead to an insertion in the posting list of i, in the Event Handler. In the case of updates, each change in the top-k list of a query leads to a deletion of the item with the lowest score, supposing that the list already contains k items. Let q be the query that is updated by an item i, and let i_k be the item that is deleted from the tok-k list of q as a result of the update. Observe that this leads to an insertion of q as a candidate of i_k in the Event Handler.

Candidate deletions can occur during event matching under two possible conditions. If a query update is detected while iterating through the candidates set, this query



(a) An item i (1) is evaluated in the Event Handler and a set of candidates is computed (2). This set is inserted in the Event Handler. The updates $U(i)$ are also retrieved (3). The insertion of i in a query q triggers the deletion of another item i' from its top-k list (4). If i' fulfills the θ_i-condition, then q is inserted as its candidate (5).



(b) An event e (1) is evaluated in the Event Handlerand the set of updates is computed (2). As before, any deleted item i' (3) can potentially be inserted in the Event Handler (4). At the same time, since i is inserted in the top-k list of q, q is no longer its candidate and has to be removed (5)

Figure 4.5: Insertion and deletions of query candidates

should be deleted, as it is no longer a candidate. A second case, is when a candidate q is found to have a minimum score higher than the θ_i threshold of the item i: This means that i needs to receive a dynamic score greater than θ_i for this query to be updated by i. If at some point, however, after the arrival of an event, i reaches a dynamic score $\mathcal{S}_{dyn}(i) > \theta_i$, due to the θ_i-condition failure, i will be re-evaluated in the Item Handler and a new, valid candidates set will be retrieved. Consequently, it is safe to remove q from the candidates set. Although deletion in both cases in not necessary for the correctness of the result, maintaining these queries as candidates would lead to an unnecessary number of false positives, deteriorating the overall event-matching performance for following event evaluations.

Figure 4.5 shows an example of the aforementioned insertions and deletions that can occur in the candidate lists.

4.3.2 Simple Event Handler

A straightforward implementation of the Event Handler posting lists is to maintain an unordered set of candidates. Assuming a dynamic array implementation of this index insertions and deletions of candidates can be performed in (amortized) constant time. On event matching, however, it is not possible to apply any early stopping condition, and all candidate queries need to be checked for updates.

Recall, however, that the candidates set computed for an item is valid for as long as the θ_i-condition holds, and potentially for a big number of arriving events. Therefore, even though the candidates posting list might contain a large number of candidates, only a small fraction of these might actually be updated after the arrival of a single event. An efficient implementation of the Event Handler index should be able to have an early stopping condition so as to avoid false positives or optimally, directly retrieve only those queries that will be updated.

4.3.3 Ordered Event Handler

Following the Fagin's Threshold Algorithm [FLN03] approach we try to define an ordering of the candidates that will allow us to define an early stopping condition and

avoid visiting all candidates in the set. Before defining the candidates order and the stopping condition, let us first discuss the update conditions.

As defined in Section 4.2 a query q is a candidate of an item i if $\mathcal{S}_{min}(q) - \mathcal{S}(q,i) \in (0, \theta_i]$. In fact, the value $\mathit{diff}(q,i) = \mathcal{S}_{min}(q) - \mathcal{S}(q,i)$ represents the dynamic score i should receive, for q to be updated by i. So, if we order candidates by the value $\mathit{diff}(q,i)$, on event arrival, we need to update the item's dynamic score and only visit candidates with a negative value of $\mathit{diff}(q,i)$. Notice that this set of visited candidates does not only guarantee that no updates will be missed: it also guarantees that the correct set of updates is retrieved without any false positives.

Nevertheless, maintaining this order is a non-trivial problem: from the moment a query is indexed as a candidate of an item i, until the moment of an event evaluation for i, the minimal scores of indexed candidates has potentially changed. The value of $\mathcal{S}_{min}(q)$ might have changed due to updates by other items or, more frequently, due to events received by their k-th item, changing the item's dynamic score and consequently the $\mathcal{S}_{min}(q)$ score of the query.

It thus becomes obvious that in a such dynamic environment it is difficult to maintain this order. In the following, we consider the consequences of maintaining the accurate order, as well as two heuristic approaches lazily re-ordering the candidates.

Exhaustive Index

In this solution, we consider accurately maintaining the order of query candidates by the score difference $\mathit{diff}(q,i)$. To achieve this, we need to make all necessary re-orderings on each $\mathcal{S}_{min}(q)$ change, independently to whether it is due to an event or an item. And these re-orderings need to be made in all postings of items that q is a candidate.

To better understand the high cost of maintenance in this index, consider that an event e arrives and is matched through the Event Handler. Since the ordering is correct, we can immediately detect the correct list of updates. However, to maintain the invariant of the index we need to find all queries q that currently contain this item i as their k-th element (determining the $\mathcal{S}_{min}(q)$). For each such query q, we need to find all items i' where q is a candidate and then re-order them. The same procedure needs to be followed also when a query q is updated by an item i.

Thus, it becomes obvious that the performance gain from avoiding false positives is outperformed by the time wasted on the order maintenance. The two following solutions follow lazy re-ordering approaches, in an effort towards achieving a good trade-off between the cost caused by false positives and the cost due to re-orderings of candidates. In both solution, we consider that for each candidate, we store an additional value representing the difference value, at the time instant of the candidates indexing.

Static-order Index

This solution considers an initial ordering of the candidates that is never altered. More precisely, after the evaluation of an item in the Item Handler, all candidates detected for this item are indexed based on the defined order: increasing order of $diff_{q,i}$. As described in the beginning of this section, additional insertions of candidates might follow this first computation of candidates, after query updates. These queries are lazily inserted in the beginning of the ordered lists, independently to the $diff_{q,i}$ value. All $\mathcal{S}_{min}(q)$ changes are also ignored. On event arrival, we need to visit all candidate queries for which the *indexed* (and not the actual) difference score is lower than the item's dynamic score, i.e. the score change of the item from the time of its last evaluation in the Item Handler.

This lazy approach allows the definition of an early stopping condition and avoids the cost of re-orderings, however due to this lack of re-orderings, the index converges towards the simple solution.

Lazy re-ordering Index

The two previous approaches either have a very high maintenance cost due to frequent re-ordering operations (Exhaustive), or an inefficient event matching due to lack of maintenance of this order (Static-order). The Lazy approach lies between these two solutions by following an "on false-positive" heuristic. Given an item i, any order changes caused by events on other items is temporarily ignored. When an event having i as target arrives, all false positives encountered until the stopping condition are re-ordered as (and if) necessary.

The intuition behind this heuristic is that the re-orderings imposed by the Exhaustive solution might never be exploited, e.g. because a candidate of an item i might be re-positioned several times before actually being visited on an event arrival. Therefore, the Lazy approach minimizes the number of re-orderings, it introduces however, a number of false positives on event matching. Given that in the general case the complexity of re-ordering an element in a sorted list has a logarithmic complexity, while the cost of a false positive, is of constant complexity, it is safe to assume that the Lazy approach is bound to be more efficient than the Exhaustive one. A false positive occurs when we check for the update condition on that is not actually updated.

4.3.4 Item Partitioning Event Handler

The main drawback of a ordered-based indexes is the high dynamicity of the $\mathit{diff}(q, i)$ metric. When considering the exhaustive index solution, we have noted that frequent re-orderings in the posting lists can lead to a rather inefficient performance. Even in the more efficient, Lazy solution, the logarithmic complexity of re-orderings can result to a poor performance in cases of a high number of false positives.

The Item Partitioning solution is based on the following observation: given two queries q_1 and q_2 that are both candidates of an item i and have both another item i' as their k-th element in the top-k list, their relative order in the candidates list of i will never change: the events received by i' or i, will change the scores $\mathit{diff}(q_1, i)$ and $\mathit{diff}(q_2, i)$ by the exact same value (the event score of arriving events) and thus, their relative order in the list remains the same. Based on this observation, in the Item Partitioning solution, we organize the query candidates of each item, with respect to their k-th element. The order assigned in each such group on their insertion remains constant for as long as their k-th element remains the same, i.e. as long as they do not receive any updates.

For instance, in Figure 4.4, item i has 5 candidate queries, two of which (q_2 and q_5) have i' as their k-th element. These queries are grouped together in a sorted list, (ordered using the *diff* function). For as long as i' remains the k-th element of both these candidate queries, the ordering of the list is static.

On event arrival, the matching operation checks each one of these groups if the

corresponding item, until reaching false negative or until the end of the list. In case of an update, however, of a query q by an item, q has to be re-indexed in a the group of its new k-th element in all items where it is a candidate. Despite the additional cost on the query update operation, as we will see later in Section 4.4, the minimization of both re-orderings (only on the case of updates) and false positives leads to an overall better performance than the previous approaches.

4.4 Experiments

In this section we give an experimental evaluation of the algorithms presented in Section 4.2 and the data structures proposed in Section 4.3, over a real dataset collected from Twitter's public streams. Through these experiments we evaluate the performance of the proposed implementations of the R^2TS algorithm against the naïve approach of the AR algorithm over a number of parameters, like the total number of stored continuous queries, and query-related (k) and scoring function-related (γ) parameters. Additionally, we assess the effect of the θ_i tuning parameter over the overall system performance. For all experiments, we present the time requirements to evaluate the items and event signals and also, the filtering achieved by each implementation over the list of candidates, as the percentage of visited queries, before meeting the stopping condition requirements.

Implementations The implementations tested in this section are the naïve event evaluation presented in algorithm AR (Naive), the simple candidates index (Simple), the lazy ordering (LazyOrder) and finally the item partitioned (ItemPart).

The implementation used for the Item Handler was a slightly modified version of the algorithm presented in [VAC12] and available as an open source library online[1]. The main functionality of item evaluation, was changed so as to additionally detect candidate queries given a value of θ_i.

[1]continuous-top-k: https://code.google.com/p/continuous-top-k/

	#items	#events	min events/item	avg events/item
DS1	10 676 097	13 787 349	1	1.29
DS5 (default)	201 581	2 013 427	5	9.99
DS10	56 417	1 105 639	10	19.60

Table 4.1: Number of items and events in each dataset

4.4.1 Experiments setup

Experiments were conducted using an Intel Core i7-3820 CPU@3.60. Algorithms were implemented in Java 7 and executed using an upper limit of 8GB of main memory (-Xmx8g). Only one core was used during execution. All times presented are the average values of 3 identical runs after an initial warm-up execution. All queries were stored before any item or event evaluation and their insertion time is not included in the results.

Datasets For all experiments, real-world datasets of items and events have been used, collected from the Twitter Stream API in a period of 5 months (from March to August 2014). From this set we have filtered out non-English tweets, as well as those without any retweets, leading to a dataset of more than than 23 million tweets and retweets (DS1). Two additional datasets, subsets of the original one, were created by only considering tweets (and corresponding retweets) with at least 5 (DS5) or 10 retweets (DS10) per item. The DS5 dataset is the default one for the experiments and contains 2.2 million retweets. In our model, an original tweet corresponds to an item and a retweet to a feedback signal, i.e. an event for the original item (Table 4.1).

Queries were generated by uniformly selecting the most frequent n-grams of 1, 2 or 3 terms from the tweet and retweet dataset leading to an average length of 1.5 terms per query.

Experimental parameters In each of the following experiments we change one parameter within a given range, while all system parameters remain constant. Table 4.2 shows the default values for each parameter, as well as the range of each parameter used for the corresponding experiments.

parameter	default value	range
#queries	900 000	[100 000, 900 000]
α	0.3	$(1-\gamma)/2$
β	0.3	$(1-\gamma)/2$
γ	0.4	$[0.05, 0.95]$
k	1	$[1, 20]$
θ_i strategy	$\theta_i^{max}/2$	$[0, 0.2]$ or $[0, \theta_i^{max}]$

Table 4.2: Experimental parameters

4.4.2 Experiment: on θ_i^{max}

Figure 4.6a experimentally confirms our theoretical results over the cost model function presented in Subsection 4.2.3. For the needs of this experiment we have assumed accurate knowledge on the maximal value of the aggregated event score (Equation 4.1) an item i will receive θ_i^{max}, but no knowledge on the number of events (r_{θ_i})it will receive. For each arriving item, we have assigned its θ_i value as a percentage of its θ_i^{max} value (horizontal axis), from 0 to 100%. The vertical axis represents the time required to match the items and events of the dataset DS5. For the NAIVE solution the θ_i value is not defined, its value is a constant line.

We can immediately observe a particular pattern in this graph: time requirements tend to be low for the values 1, 0.5, 0.33, 0.25 etc., i.e. for the values $\theta_i = \theta_i^{max}/N$, where N is a natural number. To understand this form, consider for example the case of $\theta_i = 0.5 \cdot \theta_i^{max}$. On the first evaluation of any given item i, the list of candidates with a difference up to θ_i is computed. Since θ_i corresponds to half the maximal value of the aggregated item score, the item will be re-evaluated through the Item Handler a second time, and there will be no need for a third evaluation. When a higher value is chosen, e.g. $0.6 \cdot \theta_i^{max}$, two evaluations in the Item Handler would also be required, with the difference that some extra, unnecessary candidates would be retrieved. This results in an additional cost to a) compute the unnecessary candidates and b) probably a higher number of false positives on event matching in the Event Handler.

Note that in the special case of $\theta_i = 0$ all solutions converge to the NAIVE one: $\theta_i = 0$ implies that no candidates will be maintained and thus, all event evaluations will be performed in the Item Handler, which is exactly the behavior of the NAIVE

index.

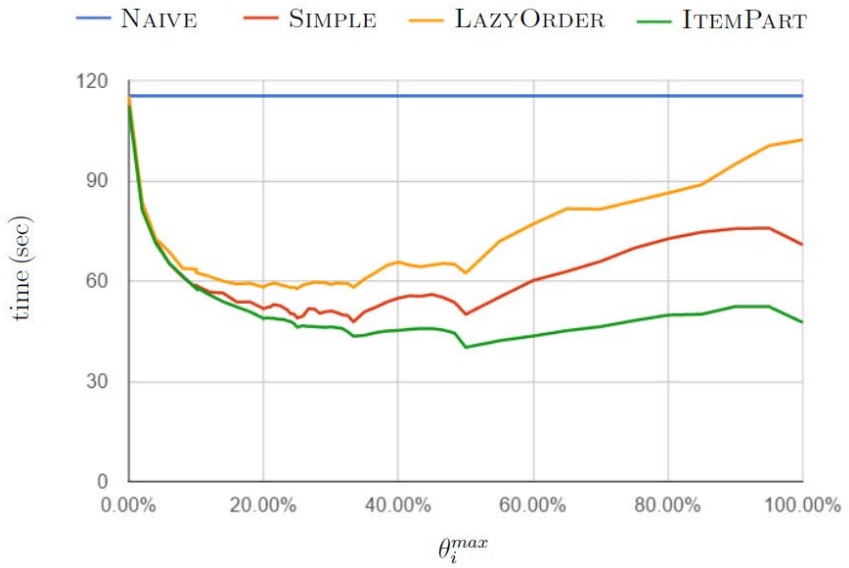
Comparing the four indexes, we can see that the ItemPart solution outperforms the other three, while the LazyOrder one has higher time requirements than the Simple, despite the definition of an early stopping condition. The relatively good performance of Simple can be explained by the lack of need for re-orderings (low maintenance cost) and the use of the dynamic vector data structure, which guarantees fast access, insertions and deletions. These factors compensate for the lack of an early stopping condition. The LazyOrder and ItemPart on the other hand, use much "heavier" data structures and need to achieve a filtering of a large portion of the candidates lists in order to outperform the Simple solution.

Figure 4.6b shows the average percentage of candidates lists visited, until the stopping condition. The Simple solution always visits 100% of the lists due to the lack of a stopping condition. Comparing this figure to the one on the time performance, we can observe that ItemPart has lower time requirements of about 20% with merely visiting 5% of the lists.

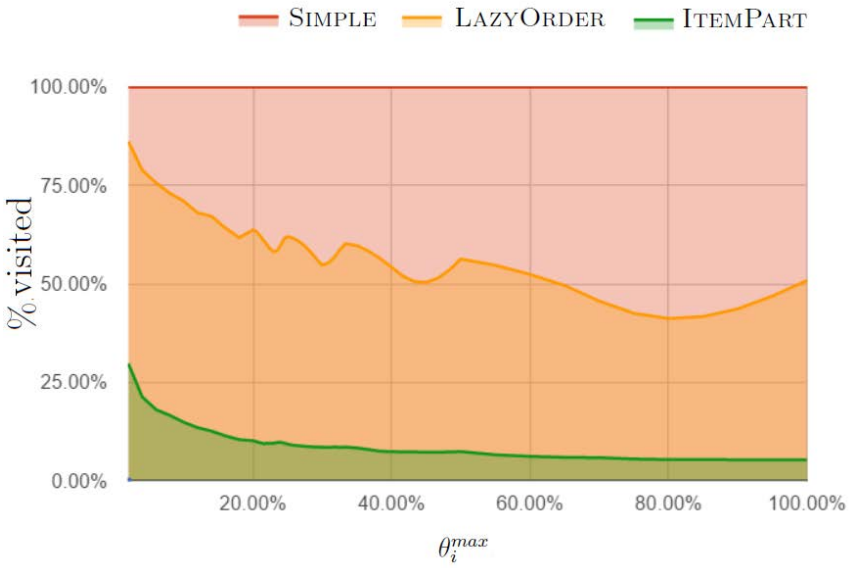
4.4.3 Experiment: on the exact value of θ_i

The previous experiment supposed a perfect estimation of the θ_i^{max} value, which is a rather difficult or even impossible task in a real-time environment. In this experiment we consider another extreme scenario: without using any knowledge or estimation on the θ_i^{max} per item, we simply assign the same θ value to all items. Figure 4.7a shows, for each value θ assigned to all items, the time required to evaluate the whole dataset (DS5).

A general observation over the indexes Simple, LazyOrder and ItemPart is that as the value of θ_i increases from 0, there is a quick improvement in the performance, while after the exceeding the optimal θ_i point, it deteriorates with a smaller slope: For very small values of θ, the candidates lists become invalidated frequently (when the θ_i-condition no longer holds) and a large number of arriving events need to be evaluated in the Item Handler. This explains the first phase where the time requirements decrease. As values of θ become much bigger, this means that the candidates lists become too big: computing the candidates is more costly and more false positives are likely to appear.



(a)



(b)

Figure 4.6: Event Handler: Experiment on θ_i^{max}

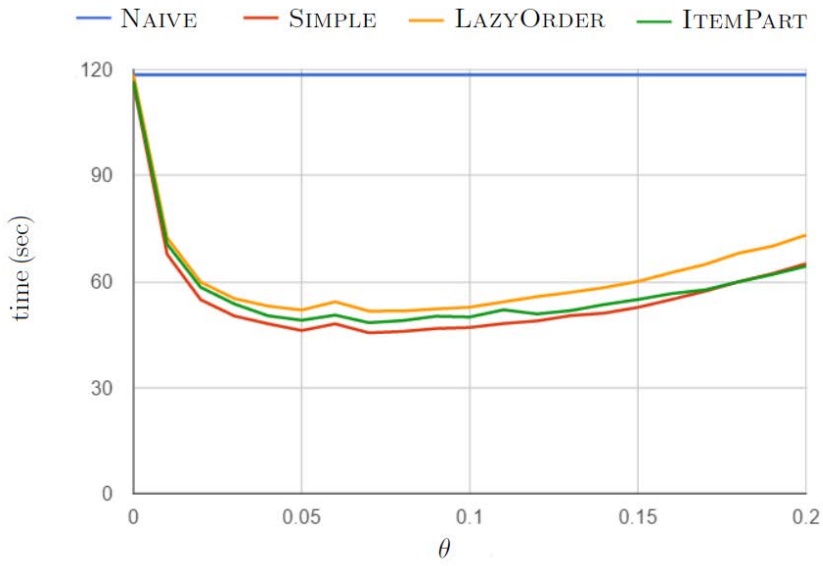
Unlike in the previous setting, here except from the NAIVE solution, the other three have similar performance with no more than 5% of difference in time requirements. This behavior is justifiable when observing the average percentage of candidates visited on each event evaluation in Figure 4.7b: in the best case for the ITEMPART an average of 10% of candidates is visited. This filtering is barely sufficient to make the ITEMPART and SIMPLE solutions have the same time requirements for the evaluation. The reason why both LAZYORDER and ITEMPART fail to achieve an earlier stopping condition and thus, a small percentage of visited queries, is due to the arbitrary assignment of the same θ value to all items. For some items this value can be large w.r.t. θ_i^{max} and thus spend unnecessary time for finding candidate queries which will never be used. On the other hand, a small value of the θ_i w.r.t. θ_i^{max} means that there will be to many costly evaluations of events in the Item Handler.

4.4.4 Experiment: on the number of queries

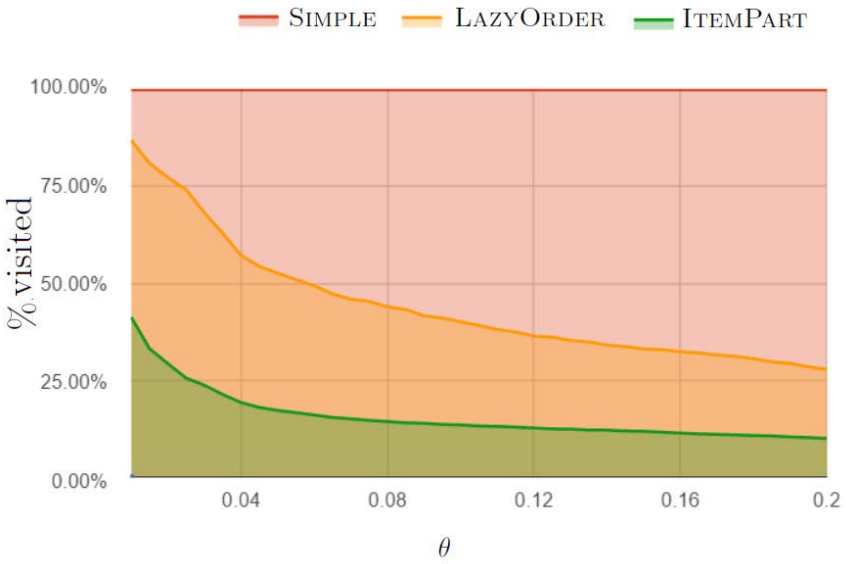
In this experiment, we test the time requirements when increasing the number of continuous queries stored in the system. All queries are inserted before evaluating items and events and their indexing time in not measured in the experiment.

From the results, presented in Figure 4.8a we can see that all four solutions scale linearly over the number of continuous queries: as more queries stored, the size of posting lists, in the Item Handler and of candidate queries lists also increases demanding more time to iterate through the indexes to retrieve the updates. Compared to the other three indexes, the ITEMPART index demonstrates better scalability: over $100,000$ queries it requires 50% of the corresponding time for the NAIVE index, while over $900,000$ it only requires 36%. The low slope of ITEMPART indicates the good performance of its stopping condition managing to filter out more candidate queries over increasing size of the list. This observation can also be made by looking at the average the percentage of candidates that are visited shown in Figure 4.8b gradually decreasing from 12 to 7%.

In terms of throughput, over 900,000 continuous queries, the ITEMPART solution would be able to handle an average of 3.2 *million* items (tweets) or events (retweets) per minute. Given that Twitter receives today an average of 340 *thousand* tweets (or



(a)



(b)

Figure 4.7: Event Handler: Experiment on a global θ

retweets) every minute our system, using a single CPU core could support an order of magnitude more tweets than Twitter actually receives. Over a total of 100,000 stored continuous queries the throughput would go to 10.9 million items and events per minute.

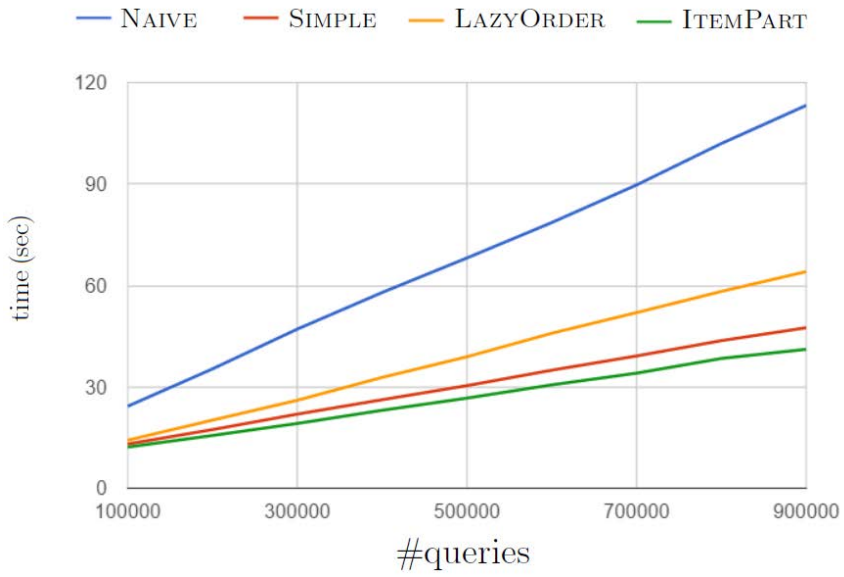
4.4.5 Experiment: on the γ value

In this experiment, presented in Figure 4.9a we test the performance of the indexes over different values of the γ parameter in our underlying scoring function presented in Equation 4.2. Recall that the γ parameter determines the importance of the aggregated event score, i.e. the dynamic part of the query-independent item score. Over small values of γ, each arriving event has minimal impact on the total score and only a small number of queries are updated by the corresponding target items of the incoming events.

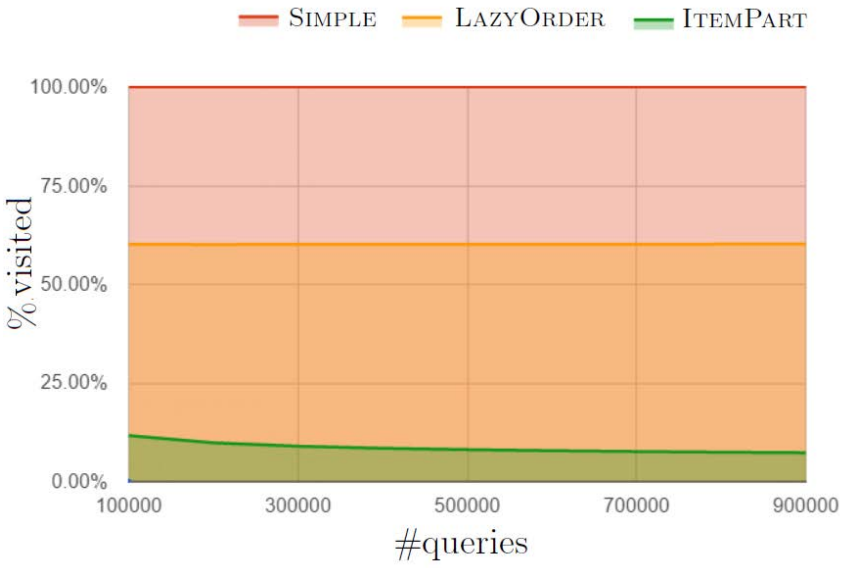
In the NAIVE index, we observe that its performance improves for values of γ up to 0.2 and then presents a constant behavior. The other three indexes on the other hand show a different behavior, with time requirement increasing for high values of γ. In fact, when γ approaches the maximal value of 1, the initial static score between queries and items is soon invalidated and the score changes due to any single event becomes more significant. As a side-effect, the candidate lists computed on item evaluation are soon invalidated as it is more likely for the indexed candidate queries minimal score to have changed. This leads to an increasing number of false positives and explains the increase in time performance. However, for the cases of values in [0.1, 0.5] (which are more likely to be applied in a real-time web system) the performance difference of any of these three systems to their optimal value is of less than 10%.

4.4.6 Experiment: on the k value

Figure 4.10a shows the performance of the four indexes over different values of the k parameter, i.e. the number of items present at any time instant in the queries' result sets. Higher values of k, result in lower value of minimal scores for the stored queries and consequently increase the number of item updates received by the queries. We can observe that all four systems scale linearly on the value of k and that the ITEMPART

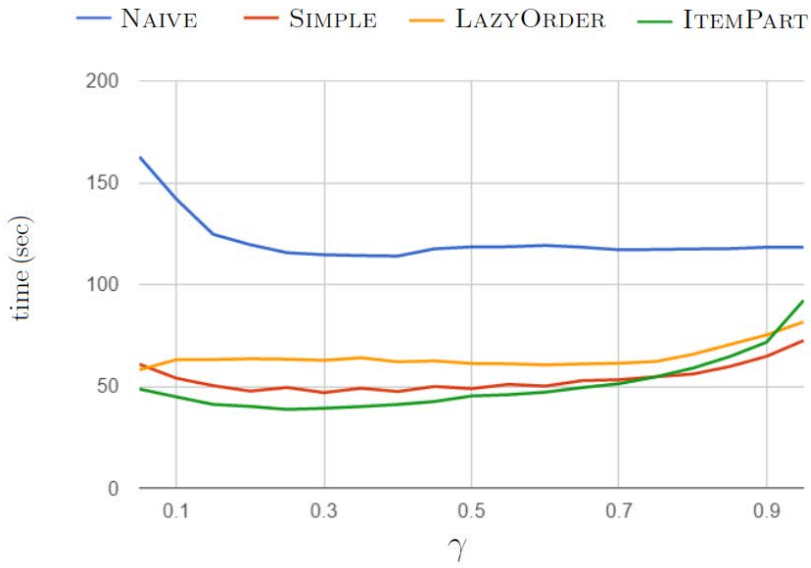


(a)

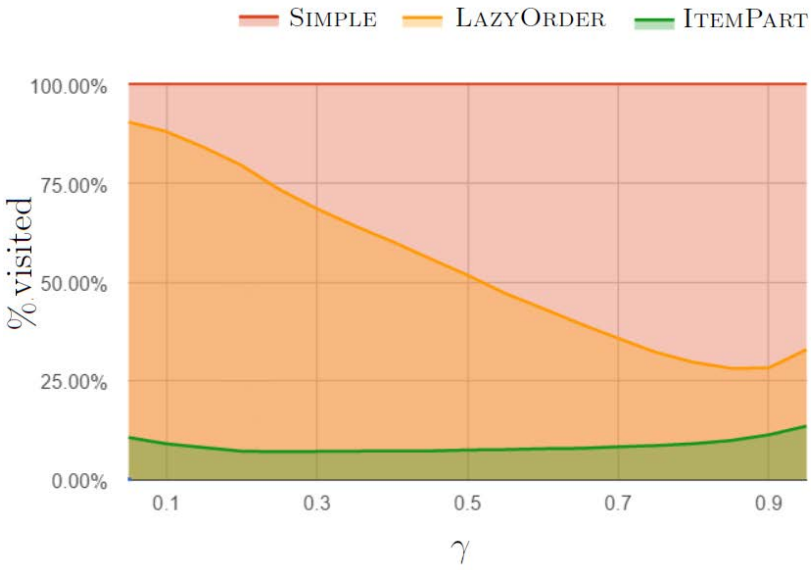


(b)

Figure 4.8: Event Handler: Experiment on the number of queries



(a)



(b)

Figure 4.9: Event Handler: Experiment on the γ parameter

and SIMPLE have a better performance of about 20% than the other two indexes when the $k = 20$.

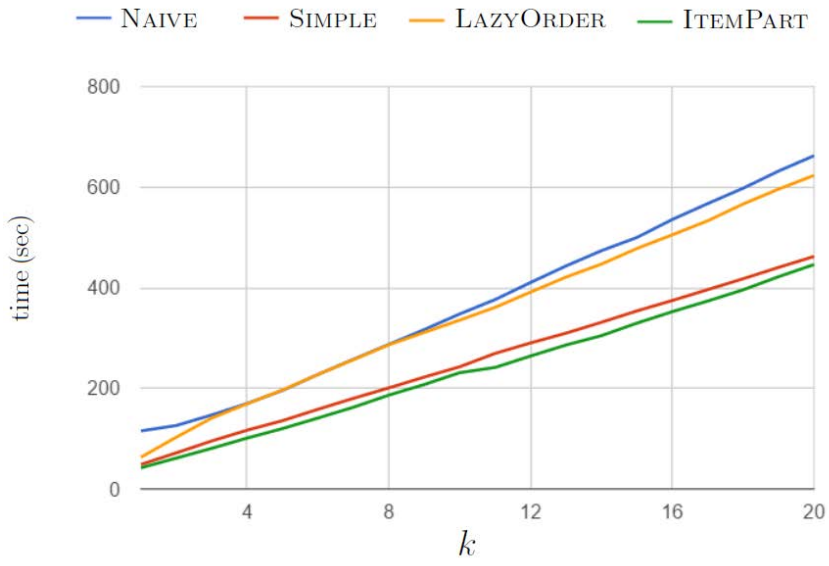
4.4.7 Experiment: on the number of events per item

In this experiment we test the overall performance of the four systems over the three datasets DS1, DS5 and DS10 (see Table 4.1). The main difference among these is the average number of events each of the items receives: DS1 has an average of only 1.29 events per item while DS10 has 19.60.

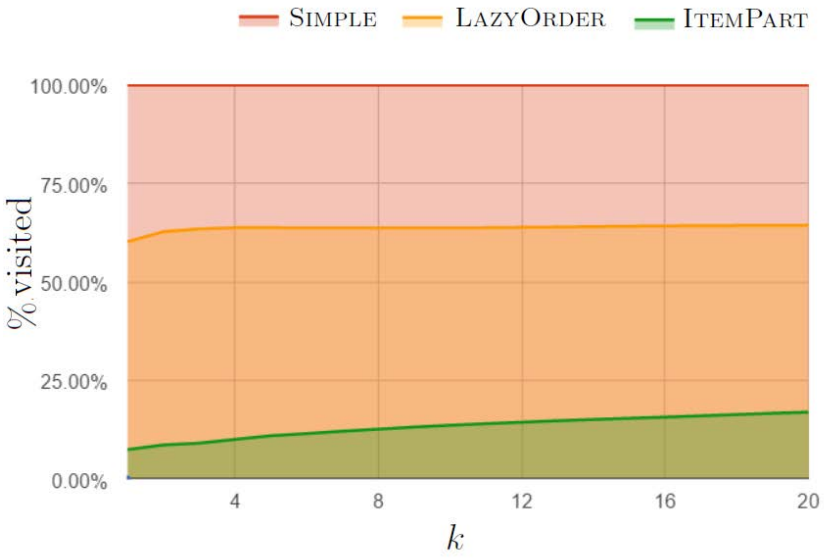
Figure 4.11 shows the average (not the total) execution time for the evaluation of each item or event in the corresponding dataset. The figure also represents the average number of updates (#updates) provoked by each of these items or events. Initially, we can observe that for all indexes the execution time increases proportionally w.r.t. the number of updates: the more queries need to be retrieved as updates, the more time is required to retrieve them. A second, more important observation is that the performance of the three indexes based on the R^2TS algorithm (SIMPLE, LAZYORDER and ITEMPART) have a better performance than the NAIVE index as the number of events per items increases. Over the DS1 dataset (with the smallest number of events/item), the ITEMPART, which is the best performing index, needs 68% of the time required by NAIVE, while over DS10 it only requires 29,8%. It thus becomes evident that our system performs better over highly dynamic environments with high numbers of events per published item.

4.4.8 Conclusions on the experiments

Our experiments demonstrate that the ITEMPART and SIMPLE solutions outperform the NAIVE and LAZYORDER ones over all settings, with ITEMPART having a slight edge over SIMPLE of about 5%. Comparing the stopping conditions, ITEMPART manages to filter with the defined stopping condition a big percentage of the candidate lists and on most cases only had to visit about 10% of the candidates before having correctly identified all updates and stopping the algorithm. However, the heavy data structures maintained for each item (an unsorted set of sorted lists) only allows ITEMPART to



(a)



(b)

Figure 4.10: Event Handler: Experiment on the k parameter

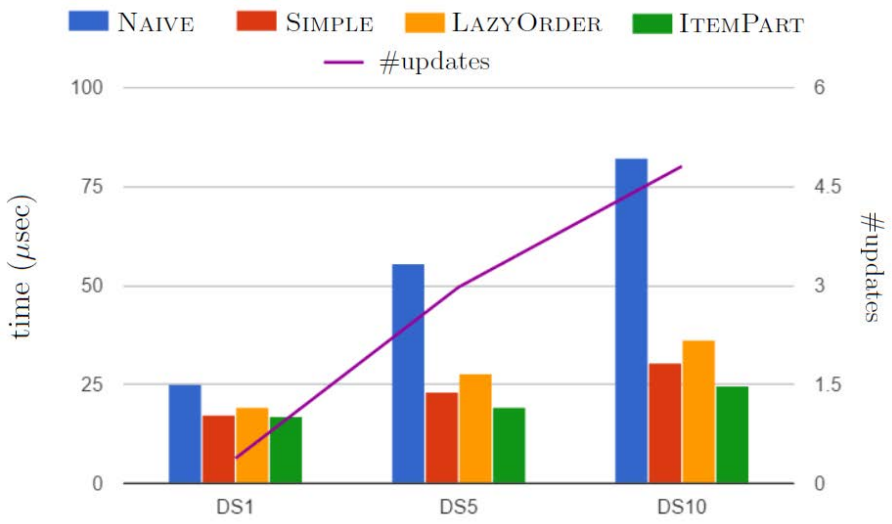


Figure 4.11: Event Handler: Experiment on the number of events per item

require from 50 to 35% the time used by the NAIVE index, which always visits all stored candidates.

4.5 Summary

In this chapter we have further generalized our proposed model of Chapter 3 by introducing a general class of dynamic scoring functions additionally considering feedback signals over real-time streams, arriving at high rates. Defining a list of candidate queries for each incoming items, we manage to efficiently evaluate incoming events as they arrive. Using an analytic model we have theoretically proven the approach to determine the optimal way of computing the candidate queries for each item. We have proposed three general categories of in-memory indexes and heuristic-based variations on these for storing the candidates and retrieving the updates triggered by events.

Our experimental evaluation shows the good performance of our proposed algorithm. As a general, important conclusion of these experiments, we can see that all three indexes based on candidate maintenance (SIMPLE, LAZYORDER and ITEMPART) achieve a high throughput of items and events, with ITEMPART managing to handle

3.2 million signals in a single minute, over 900 thousand stored continuous queries. Finally, we have shown that our solutions are much faster w.r.t. the naïve solution over high loads of events. These experimental results demonstrate the great efficiency of our R^2TS algorithm and the proposed solutions, which make them capable of answering continuous queries over highly dynamic environments with real-time web information.

5

MeowsReader: A Feedback Enabled Ranking & Filtering Prototype

This chapter presents MeowsReader, a real-time news ranking and filtering prototype, putting together the outcomes of our work presented in Chapter 3 and Chapter 4. Users express their interests by simple text queries and continuously receive the best matching results, deriving from U.S. news media, in an alert-like environment. Additionally, a trend detection mechanism automatically generates trending entities from the input streams, which can smoothly be added to user profiles in form of keyword queries.

Our prototype demonstrates that real-time search, considering news media information coupled with social media feedback, can be formally defined and efficiently implemented using an expressive framework supporting *continuous top-k queries* with *generalized scoring functions* as presented in Chapter 3 and Chapter 4. The key component that differentiates MeowsReader from existing real-time search systems based on periodic query evaluation (e.g. Twitter Search), is the *top-k filtering* module which *continuously* processes the incoming stream of items, through the Item Handler (see Chapter 3) and feedback signals, through the Event Handler (see Chapter 4) for immediately identifying for each new item or signal the relevant top-k query results that have to be updated.

The MeowsReader prototype continuously aggregates news articles from more than 1,200 RSS feeds deriving from U.S. news media sources, retrieving on average 50 newly published articles per minute. At the same time MeowsReader collects micro-blog posts (tweets) from Twitter using the public Streaming[1] and Search[2] APIs. Among the retrieved tweets, those that contain links towards our underlying news sources serve as feedback signals on the corresponding news articles. Additionally, clicks of MeowsReader users on articles are also used as a source of positive feedback. Modeling news articles as *items* and both tweets and clicks as *events*, MeowsReader employs our proposed functionality and architecture (Figure 5.1) and offers real-time filtering and ranking of incoming information for thousands of stored continuous queries.

[1] dev.twitter.com/streaming/
[2] dev.twitter.com/rest/public/search

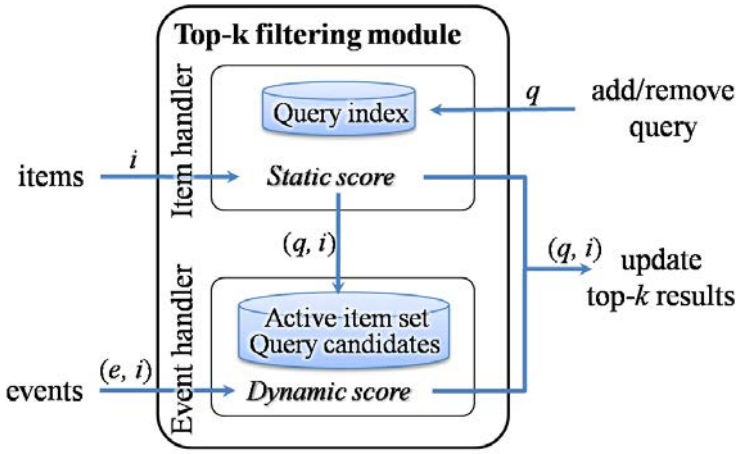


Figure 5.1: Real-time search with user feedback

The rest of this Chapter is organized as follows. Section 5.1 we present the general framework adapted by MeowsReader. Section 5.2 presents the architecture and main components of this prototype. Section 5.3 presents the User Interface of our application and finally, Section 5.4 summarizes the main outcomes of this work.

The prototype presented in the chapter has been published in [VAC14] and an online demonstration of MeowsReader is accessible at http://gateway.lip6.fr:8080/meows.

5.1 Framework

The MeowsReader demonstration implements our *continuous top-k query* framework for real-time search over multiple Web 2.0 textual streams. It simultaneously handles three streams of input: queries, items and events. The queries stream contains information on the insertion and deletion of continuous text queries and immediately updates the query index maintained in the Item Handler. Notice however, that due to the lack of a Query Handler component (see Figure 1.3) the inserted queries are not immediately evaluated and their result sets are initially empty. Incoming items, arriving from more than a thousand sources, are merged into a single streams and are linearly treated in the Item Handler as explained in Chapter 3. Event streams, are already processed clicks and tweets concerning a single item and are handled in the Event Handler (see Chapter 4).

A core component of MeowsReader is our underlying scoring function presented in Section 4.1 and in the Equation 4.2 that takes into account query-dependent text similarity (cosine similarity), static (article's source authority) and dynamic (tweets and clicks) query-independent item importance and recency of information using an exponential decay function.

The main goal of the *top-k filtering module* is to detect for each new incoming item or event the *top-k query results* to be updated. Figure 5.1 illustrates the general idea of the overall query processing model. The *Item Handler* processes all incoming items i and immediately detects all queries q which have to be updated according to the static similarity and static item score (the dynamic item score is by definition equal to 0 at item arrival). The *Event Handler* processes incoming events, i.e. tweets or clicks concerning a given item, and continuously decides if a query result has to be updated because of the corresponding item score change.

5.2 The MeowsReader architecture

In this section we describe the architecture of MeowsReader, a complete Web 2.0 news aggregation prototype featuring non-homogeneous scoring functions which can take account of social media focus and user feedback streams for item filtering and ranking.

The overall system architecture of MeowsReader relies on a Publish/Subscribe interaction scheme as presented in Figure 5.2. On the back-end, the system collects *information items* and *feedback signals* as described in Section 5.1. The *Stream aggregation* module crawls an extensible collection of RSS/Atom feeds published in online media sources and generates a unique stream of information items i. Feedback signals are collected by the *Feedback manager* module from the real-time web (Twitter) and by capturing user clicks on MeowsReader interface. Among the collected tweets, we consider in particular those containing a link towards news items that are already registered in the system. These tweets represent feedback signals on items and cause the dynamic item score ($\mathcal{S}_{dyn}(i)$ in Equation 4.2) to increase. The score change is a linear combination of the tweet related information (retweets, favorites) and author information (number of followers) similar to [CLOW11].

The front-end of the system is composed of a *Web user interface* and the *Subscrip-*

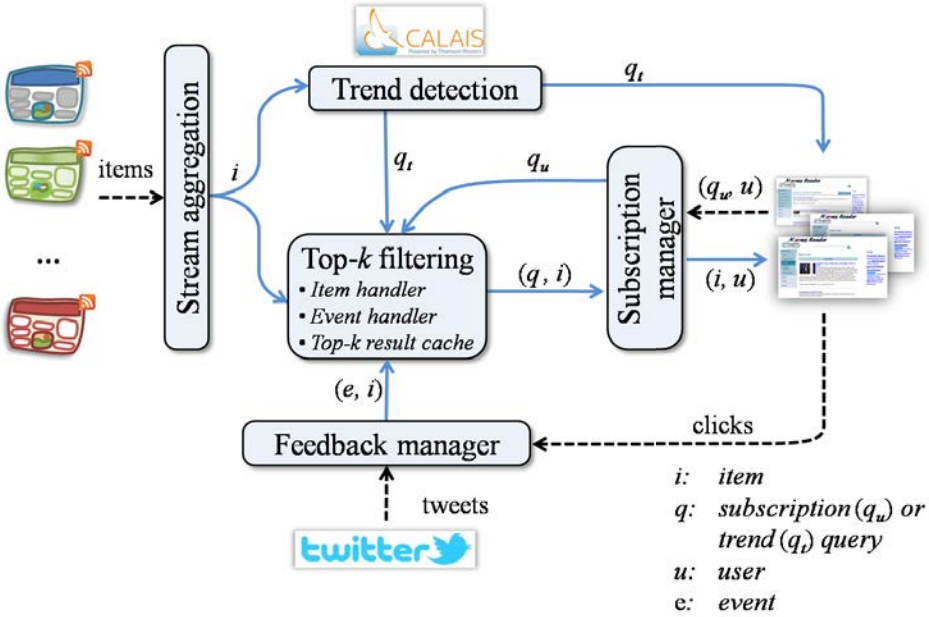


Figure 5.2: MeowsReader Architecture

tion manager. The Web user interface (see the Appendix page and the online demonstration) enables users to express and register their interest on incoming information items through *keyword queries*. It also collects user click behavior for the *Feedback manager*. The Subscription manager module takes care of the interaction between the *top-k filtering* module and the user interface.

New items are simultaneously processed by the *Top-k filtering* module (Figure 5.1) and the *Trend detection* module which extracts entities from the contents of items and detects those entities whose frequency in recent items has a bursty behavior. A certain number of such "popular" entities are transformed into *trend queries* q_t, automatically registered in the top-k *Query index* (Figure 5.1) and proposed to users through the Web user interface (see below for more details).

All the core components of MeowsReader are implemented in Java 7. The user interface is implemented using JSP, javascript and general HTML 5 technologies, like WebSockets, permitting the immediate update of the users web page in cases of notifications on item updates.

5.2.1 Top-k filtering

A major feature of our framework is the tight and efficient integration of a continuous *top-k filtering* technique with a real-time web based ranking approach (see Section 5.1). The implementation of this core component is an extension of the solutions presented in Chapter 3 [VAC12][3] and a simplified version of the algorithms presented in Chapter 4. Its main role is to match new items against queries which can be trends, user subscriptions or both. The generated result is a stream of query-item pairs (q, i) where i is added to the top-k result of q. *All* subscriptions to a given query simultaneously generate the same stream which is pushed to the Subscription Manager module for user notification. All detected query-item pairs are maintained in a *top-k result cache* for immediately generating a result for new subscriptions to existing user-defined or system generated trend queries. This architecture simplifies the whole process and in particular the query processing task which becomes independent of the number of user subscriptions.

5.2.2 Trend detection

Trend detection is an important feature of online Web 2.0 media applications. The goal of this module is to automatically generate potential future subscriptions reflecting the contents of recently published information. The usage of trends is twofold. Trends indicate to users important news that might interest them and that they could potentially miss if they only observe a static set of subscriptions. What is more, once detected, trends are handled by the system as if they were user subscriptions and as such, matched item results are maintained in the top-k cache. Thus, trend queries always generate non-empty results and simulate ad-hoc query behavior (observe that any other *new* user-defined query starts from an initially empty stream of results). In MeowsReader we apply a straightforward trend detection mechanism based on named entity frequency statistics. MeowsReader uses the OpenCalais [4] service for extracting semantic entity phrases from incoming item contents. MeowsReader presents as trend queries those named entity phrases, whose terms show the most *bursty behavior*. Bursty

[3]The source code is available online at http://code.google.com/p/continuous-top-k
[4]http://www.opencalais.com/

behavior of a term is detected by comparing the relative difference between the *total* term frequency observed so far and the *recent* term frequency observed in a sliding window over the input item stream.

5.3 MeowsReader user interface

MeowsReader is based on a large collection of more than a thousand general and specialized U.S. news media RSS feeds and from related feedback continuously collected using Twitter Search and Stream APIs. Users can directly access the online MeowsReader application using the MeowsReader *Web user interface* on any Javascript enabled web browser client. They can immediately access and explore a representative collection of trends related to events taking place at that moment. In addition, users can subscribe to queries defined in an ad-hoc manner or by refining existing trend queries. User interaction with the platform, i.e. clicks, are registered by the system and influences the dynamic item score values. In order get more insights about the functionality, the collected feedback signals and other statistics are visible through the user interface.

Figure 5.3 shows a screenshot of the MeowsReader user interface. It is decomposed into three frames which provide access to all broker services. In the main central frame, users can see a personalized view of a selection of the most highly ranked recent articles produced by all their subscriptions. Users can view and edit these subscriptions on the left frame, where subscriptions are categorized into topics (e.g. Politics, Sports, ...) for facilitating subscription and result monitoring. Selecting a topic or a subscription generates a specialized view of top-k results in the main frame. On the right frame, MeowsReader displays a random subset of the current trends as sample subscriptions, which the user can accept, edit or remove.

All user categories, subscription and other preferences are locally stored in the browser. This local storage and the usage of a low-level web socket protocol without the necessity of cookies protects privacy as no personal data are exchanged with the server and selected subscriptions are only known to the server while the user stays online.

Figure 5.3: The MeowsReader User Interface: Users can search through keyword-based queries (top), manage their existing subscriptions (left), view articles as ranked and filtered by the system (center) and view trending entities (right).

5.4 Summary

In this Chapter we have presented MeowsReader, our Java-implemented news reader, aggregating news articles form more than 1,200 U.S. online sources and filtering them using the continuous top-k query approach. An additional stream of feedback signals, deriving from Twitter and from clicks on the online MeowsReader website, is exploited to improve quality of results at real-time. We have additionally implemented a trend detection module, discovering trending entities and proposing them to users as potential subscriptions.

Overall, this work provides a second working application of our proposed general framework on continuous queries. While on Chapter 4 we have considered tweets as items and re-tweets as feedback signals here, published news articles represent the information items to be ranked, while tweets (and retweets, favorites) and also clicks represent the feedback events.

Figure 5.4: As new item-updates become available for a user's existing subscription, a notification appears (yellow circle). By selecting the subscription, the user can see the updates (shown by the arrow).

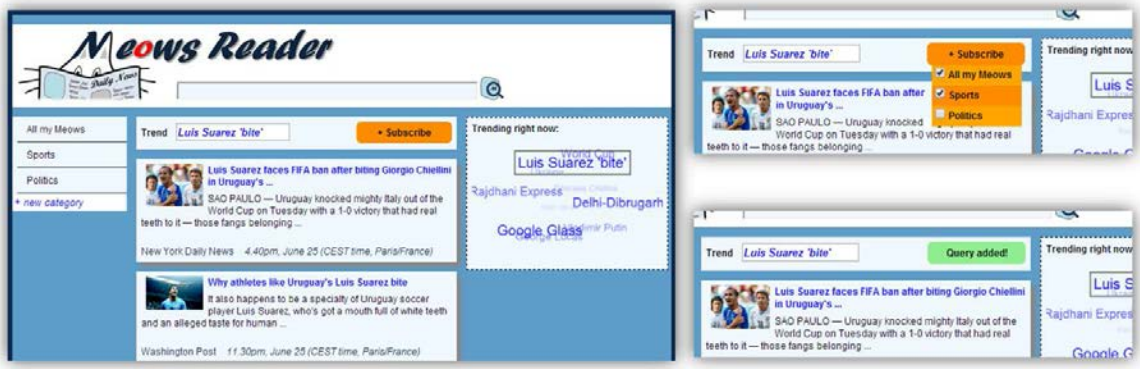


Figure 5.5: A user can select any trend, which immediately displays the cached result. Trends can also be modified and added to one or more user defined subscription categories.

Figure 5.6: The MeowsReader Sandbox User Interface: users can compare the result of different score function configurations. As an example, for query "Barack Obama" Strategy 1 (Social media attention) promotes the dynamic item scores (high γ parameter), while Strategy 2 (Information recency) changes the decay function and consequently favors more recent items in the top-k result.

6

Conclusions

6.1 Research Summary

In this thesis, we have studied the problem of continuous query filtering over dynamic text streams. Given the massive amounts of user generated contents being published in Web 2.0 applications, in conjunction with content from traditional information producers, such as online news media, accessing information becomes a challenging task. In this context, our goal is to achieve effective, *near real-time information awareness for millions of users*.

To filter such streams of information, we employ the *continuous top-k query evaluation approach*. Users issue keyword queries, which are used to provide personalized views over the streams of items (e.g. of news articles or micro-blog messages). An underlying scoring function estimates the importance of items in the stream w.r.t. queries. The problem of continuous top-k query evaluation is defined as the *maintenance of the k most highly ranked items w.r.t a given (continuous) query at each time instant*. In our proposed model, we have employed a general class of scoring functions, considering *query-dependent* content relevance (e.g. with cosine similarity or Okapi BM25), *query-independent* item importance (considering parameters such as information novelty, diversity, feedback signals etc.) and also taking into account *freshness* of information (using aging models and decay functions).

Existing solutions on continuous top-k query evaluation assume the use of scoring functions that cannot support any dynamic or static query-independent ranking parameters. On the other hand, works on the real-time web which do consider highly dynamic scoring functions over long-running queries, fail to handle the streams in an online way and follow, instead, an approach of periodic execution of snapshot queries.

Our main contribution through this thesis is the definition of novel algorithms and data structures for the efficient processing of continuous top-k text queries over highly dynamic streams of items and feedback signals. Our experiments, conducted over real collections of news media and social media data, show that our proposed solutions are scalable and can be applied in the highly demanding context of real-time web streams.

MeowsReader, our publicly available news recommendation prototype, demonstrates an example application of our approach which, using minimal resources, is capable of filtering thousands of news article per day for thousands of stored queries over dynamic feedback streams of clicks and real-time web streams.

6.2 Perspectives

6.2.1 Employing dynamic continuous queries

In our work we have considered that the stored continuous queries are user defined and remain *static*. In practice, however, most modern commercial search engines, like the Google search engine or Twitter search, additionally consider user profiles when answering queries, i.e. users past interaction with the system (e.g. clicks) or their social network (e.g. followees in Twitter) [BGL+12]. The use of social and news media by millions of users and the increasing amount of time spent by each user in such online systems makes user profiles highly dynamic and thus, their continuous evaluation over streams of information becomes an non trivial problem.

In our proposed problem formulation, user profiles could be represented by continuous queries which evolve in time. The evolution of a continuous query might be seen as a replacement of an old version by a new one or as a modification of its definition (insertion/deletion of terms and/or change of term weights). In both cases, the query index structure we proposed in Chapter 3 should be adapted to accommodate these updates. In particular, indexed query points then might move "vertically" in the two dimensional inverted query index of a given term. While our current implementations do not exclude this kind of updates, they might require new optimizations for avoiding performance loss for high update frequency.

Spatial queries is another motivating example for supporting dynamic queries using the continuous top-k evaluation approach. For instance, Google Now[1], an application recommending to mobile users popular places, traffic and public transport updates, breaking news etc. bases its results on the location of users additionally to their profiles.

[1] www.google.com/landing/now/

To the best of our knowledge, there are no existing solutions following the continuous top-k query evaluation approach that consider such dynamic user profiles.

6.2.2 Internet of things

During the past decade, we have witnessed the transformation of the web from a static environment to a vibrant information place. We are nowadays traversing the era of Web 2.0 and of the real-time web, with massive amounts of user-generated information and feedback becoming available at high rate streams. So, one could wonder: what is next?

The vision of the *Internet of Things* (IoT) [Kop11] lies in enabling objects, such as health monitor devices, drones[2], smart watches or even simple lamps, to be connected on the Internet and be able to transmit information. In information filtering, IoT opens yet another dimension on content generation. It enables a real-time monitoring and sensing of the physical world and not only the digital world [CP15]. Moreover, it allows the construction of far more dynamic user profiles for real-time context aware recommendation [Gup15] and data filtering [FGL+13]. Research questions that rise in this context, include the modeling of information filtering in such novel environments and providing efficient, scalable and personalized filtering in real-time.

The framework we proposed in this thesis combines queries, information items and feedback signals. In this setting, users are interested in personalizing massive information flows via continuous queries. Our underlying assumption is that textual items are produced and consumed by users, while their diffusion is constantly assessed by a large community through shares, likes, etc. In a different context, IoT information ranges from raw data produced by sensors (e.g. measurements) to high level data produced by analytic applications (e.g. events) while user feedback becomes more and more implicit regarding items (e.g., users attention can be inferred by the duration of the activities involving an item or the observed emotional affective state, etc.), of both the digital and physical worlds.

We argue that current state of the art techniques mainly based on the extensive storage of all data and the execution of snapshot queries are not a viable solution in the

[2]Drones: unpiloted aerial vehicles, remotely controlled

context of real-time IoT applications. Our work lays the foundation for an alternative continuous filtering and data processing architecture able to manage the traditional trade-off between expressivity and performance, based on new indexing and pruning techniques for reducing memory and processing costs. One direction of future work is to generalize the proposed continuous top-k filtering model to other kinds of data sharing between humans and machines, where scoring functions and real-time filtering play a crucial role. Another perspective is to study dynamic scoring functions for complex event processing as a common filtering functionality of real-time analytics applications in Internet of Things (IoT) [PZCG14, GSJM14].

Bibliography

[AAYI+13] Sofiane Abbar, Sihem Amer-Yahia, Piotr Indyk, Sepideh Mahabadi, and Kasturi R. Varadarajan. Diverse near neighbor problem. In *Proceedings of the Twenty-ninth Annual Symposium on Computational Geometry*, SoCG '13, pages 207–214, New York, NY, USA, 2013. ACM.

[AK11] Albert Angel and Nick Koudas. Efficient diversity-aware search. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 781–792, New York, NY, USA, 2011. ACM.

[AVB15a] Abdulhafiz Alkhouli, Dan Vodislav, and Boris Borzic. Algorithms for continuous top-k processing in social networks. In *1st International Symposium on Web AlGorithms*, 2015.

[AVB15b] Abdulhafiz Alkhouli, Dan Vodislav, and Boris Borzic. Continuous top-k processing of social network information streams: a vision. In *In ISIP 2014 post proceedings*, Springer 2015, 2015.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[BGL+12] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1360–1369, April 2012.

[BVW04] Klaus Berberich, Michalis Vazirgiannis, and Gerhard Weikum. T-rank: Time-aware authority ranking. 2004.

[CATV11] Jordi Creus, Bernd Amann, Nicolas Travers, and Dan Vodislav. Roses: A continuous content-based query engine for rss feeds. In Abdelkader

Hameurlain, StephenW. Liddle, Klaus-Dieter Schewe, and Xiaofang Zhou, editors, *Database and Expert Systems Applications*, volume 6861 of *Lecture Notes in Computer Science*, pages 203–218. Springer Berlin Heidelberg, 2011.

[CDCS10] Mario Cataldi, Luigi Di Caro, and Claudio Schifanella. Emerging topic detection on twitter based on temporal and social terms evaluation. In *Proceedings of the Tenth International Workshop on Multimedia Data Mining*, MDMKDD '10, pages 4:1–4:10, New York, NY, USA, 2010. ACM.

[CKP04] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Querying imprecise data in moving object environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(9):1112–1127, 2004.

[CLOW11] Chun Chen, Feng Li, Beng Chin Ooi, and Sai Wu. Ti: An efficient indexing mechanism for real-time search on tweets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 649–660, New York, NY, USA, 2011. ACM.

[CP15] Vassilis Christophides and Themis Palpanas. Report on the first international workshop on personal data analytics in the internet of things (pda@ iot 2014). *SIGMOD Record*, 44(1), 2015.

[CSSX09] Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu. Forward decay: A practical time decay model for streaming systems. In *IEEE International Conference on Data Engineering (ICDE'09)*, pages 138–149, 2009.

[DCGR05] Gianna M. Del Corso, Antonio Gullí, and Francesco Romani. Ranking a stream of news. In *Proceedings of the 14th International Conference on the World Wide Web (WWW'05)*, pages 97–106, 2005.

[DDGR07] Abhinandan S Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, pages 271–280. ACM, 2007.

[DFMGL12] Gianmarco De Francisci Morales, Aristides Gionis, and Claudio Lucchese. From chatter to headlines: Harnessing the real-time web for personalized news recommendation. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, WSDM '12, pages 153–162, New York, NY, USA, 2012. ACM.

[DK68] AP Dempster and R.M. Kleyle. Distributions determined by cutting a simplex with hyperplanes. *The Annals of Mathematical Statistics*, pages 1473–1478, 1968.

[DSP09] Marina Drosou, Kostas Stefanidis, and Evaggelia Pitoura. Preference-aware publish/subscribe delivery with diversity. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 6:1–6:12, New York, NY, USA, 2009. ACM.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31:109–118, 2002.

[FGL+13] Z.T. Feng, Y. Ge, C. Liu, W. Lu, B. Yang, and Q. Yu. Data filtering in the internet of things, October 31 2013. US Patent App. 13/869,236.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[GDH04] Evgeniy Gabrilovich, Susan Dumais, and Eric Horvitz. Newsjunkie: providing personalized newsfeeds via analysis of information novelty. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04)*, pages 482–490, 2004.

[GNOT92] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using collaborative filtering to weave an information tapestry. *Commun. ACM*, 35(12):61–70, December 1992.

[GSJM14] Nithyashri Govindarajan, Yogesh Simmhan, Nitin Jamadagni, and Prasant Misra. Event processing across edge and the cloud for internet of things applications. In *Proceedings of the 20th International Conference on Management of Data*, COMAD '14, pages 101–104, Mumbai, India, India, 2014. Computer Society of India.

[Gup15] B. Gupta. Real-time context aware recommendation engine based on a user internet of things environment, January 15 2015. US Patent App. 14/324,917.

[HAA12] Roxana Horincar, Bernd Amann, and Thierry Artires. Online change estimation models for dynamic web resources. In Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, editors, *Web Engineering*, volume 7387 of *Lecture Notes in Computer Science*, pages 395–410. Springer Berlin Heidelberg, 2012.

[HLLM06] Yang Hu, Mingjing Li, Zhiwei Li, and Wei-ying Ma. Discovering authoritative news sources and top news stories. In *Proceedings of the Third Asia conference on Information Retrieval Technology (AIRS'06)*, pages 230–243, 2006.

[HMA10] Parisa Haghani, Sebastian Michel, and Karl Aberer. The gist of everything new: personalized top-k processing over web 2.0 streams. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management (CIKM'10)*, pages 489–498, 2010.

[HMA12] Parisa Haghani, Sebastian Michel, and Karl Aberer. Efficient monitoring of personalized hot news over web 2.0 streams. *Computer Science - Research and Development*, 27:81–92, 2012.

[Hom12] H. Homei. Two-sided power random variables. *arXiv preprint arXiv:1206.1998*, 2012.

[HVT+11] Zeinab Hmedeh, Nelly Vouzoukidou, Nicolas Travers, Vassilis Christophides, Cedric du Mouza, and Michel Scholl. Characterizing web syndication behavior and content. In *Proceedings of the 12th*

International Conference on Web Information System Engineering (WISE'11), LNCS, pages 29–42, 2011.

[JR83] Carlo Jacoboni and Lino Reggiani. The monte carlo method for the solution of charge transport in semiconductors with applications to covalent materials. *Reviews of Modern Physics*, 55(3):645, 1983.

[Kar01] George Karypis. Evaluation of item-based top-n recommendation algorithms. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 247–254, New York, NY, USA, 2001. ACM.

[Kop11] Hermann Kopetz. Internet of things. In *Real-Time Systems*, Real-Time Systems Series, pages 307–323. Springer US, 2011.

[KPH04] Dmitri Kalashnikov, Sunil Prabhakar, and Susanne Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15:117–135, 2004.

[KSJ09] Ioannis Konstas, Vassilios Stathopoulos, and Joemon M. Jose. On social networks and collaborative recommendation. In *Proceedings of the 32Nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, pages 195–202, New York, NY, USA, 2009. ACM.

[LBLT15] Yuchen Li, Zhifeng Bao, Guoliang Li, and Kian-Lee Tan. Real time personalized search on social networks. ICDE, 2015.

[LDP10] Jiahui Liu, Peter Dolan, and Elin Rønby Pedersen. Personalized news recommendation based on click behavior. In *Proceeding of the 14th international conference on Intelligent user interfaces (IUI'10)*, pages 31–40, 2010.

[LS03] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 129–140, 2003.

[LYWL05] Xuemin Lin, Yidong Yuan, Wei Wang, and Hongjun Lu. Stabbing the sky: efficient skyline computation over sliding windows. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pages 502–513, April 2005.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD'06)*, pages 635–646, 2006.

[MC10] Xi Mao and Wei Chen. A method for ranking news sources, topics and articles. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 4, pages 170–174, 2010.

[MK10] Michael Mathioudakis and Nick Koudas. Twittermonitor: Trend detection over the twitter stream. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 1155–1158, New York, NY, USA, 2010. ACM.

[MLY+10a] Yajie Miao, Chunping Li, Liu Yang, Lili Zhao, and Ming Gu. Evaluating importance of websites on news topics. In *PRICAI 2010: Trends in Artificial Intelligence*, volume 6230 of *LNCS*, pages 182–193, 2010.

[MLY+10b] Yajie Miao, Chunping Li, Liu Yang, Lili Zhao, and Ming Gu. Evaluating importance of websites on news topics. In *PRICAI 2010: Trends in Artificial Intelligence*, volume 6230 of *LNCS*, pages 182–193, 2010.

[MME+14] A. Magdy, M.F. Mokbel, S. Elnikety, S. Nath, and Yuxiong He. Mercury: A memory-constrained spatio-temporal real-time search on microblogs. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 172–183, March 2014.

[MP09] Kyriakos Mouratidis and HweeHwa Pang. An incremental threshold method for continuous text search queries. In *IEEE International Conference on Data Engineering (ICDE'09)*, pages 1187–1190, 2009.

[MP11] Kyriakos Mouratidis and HweeHwa Pang. Efficient evaluation of continuous text search queries. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23:1469–1482, 2011.

[MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[MSN11] Enrico Minack, Wolf Siberski, and Wolfgang Nejdl. Incremental diversification for very large sets: a streaming-based approach. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 585–594. ACM, 2011.

[MZL+11] Hao Ma, Dengyong Zhou, Chao Liu, Michael R. Lyu, and Irwin King. Recommender systems with social regularization. In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pages 287–296, New York, NY, USA, 2011. ACM.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. (1999-66), November 1999. Previous number = SIDL-WP-1999-0120.

[PZA08] Kresimir Pripuzic, Ivana Podnar Zarko, and Karl Aberer. Top-k/w publish/subscribe: Finding k most relevant publications in sliding time window w. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 127–138, New York, NY, USA, 2008. ACM.

[PZCG14] Charith Perera, Arkady Zaslavsky, Peter Christen, and Dimitrios Georgakopoulos. Context aware computing for the internet of things: A survey. *Communications Surveys & Tutorials, IEEE*, 16(1):414–454, 2014.

[RCCT14] Weixiong Rao, Lei Chen, Shudong Chen, and Sasu Tarkoma. Evaluating continuous top-k queries over document streams. *World Wide Web*, 17(1):59–83, 2014.

[RWJ+95] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. Okapi at trec-3. *NIST SPECIAL PUBLICATION SP*, pages 109–109, 1995.

[SB88] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[SGFJ13] Alexander Shraer, Maxim Gurevich, Marcus Fontoura, and Vanja Josifovski. Top-k publish-subscribe for social annotation of news. In *Proceedings of the 39th International Conference on Very Large Data Bases*, 2013.

[SH09] AR Soltani and H. Homei. Weighted averages with random proportions that are jointly uniformly distributed over the unit simplex. *Statistics & Probability Letters*, 79(9):1215–1218, 2009.

[SJLL00] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 331–342, 2000.

[SOM10] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: Real-time event detection by social sensors. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 851–860, New York, NY, USA, 2010. ACM.

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, pages 321–330, New York, NY, USA, 1992. ACM.

[TP06] Yufei Tao and Dimitris Papadias. Maintaining sliding window skylines on data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 18(3):377–391, March 2006.

[VA87] W. Van Assche. A random variable uniformly distributed between two independent random variables. *Sankhyā: The Indian Journal of Statistics, Series A*, pages 207–211, 1987.

[VAC12] Nelly Vouzoukidou, Bernd Amann, and Vassilis Christophides. Processing continuous text queries featuring non-homogeneous scoring functions. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 1065–1074, New York, NY, USA, 2012. ACM.

[VAC14] Nelly Vouzoukidou, Bernd Amann, and Vassilis Christophides. Meowsreader: Real-time ranking and filtering of news with generalized continuous top-k queries. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, pages 2066–2068, New York, NY, USA, 2014. ACM.

[WLXX13] Lingkun Wu, Wenqing Lin, Xiaokui Xiao, and Yabo Xu. Lsii: An indexing structure for exact real-time search on microblogs. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 482–493, April 2013.

[WZRM08] Canhui Wang, Min Zhang, Liyun Ru, and Shaoping Ma. Automatic online news topic ranking using media focus and user attention based on aging theory. In *Proceeding of the 17th ACM Conference on Information and Knowledge Management (CIKM'08)*, pages 1033–1042, 2008.

[YGM94] Tak W. Yan and Hector Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19(2):332–364, June 1994.

[YGM99] Tak W. Yan and Hector Garcia-Molina. The sift information dissemination system. *ACM Trans. Database Syst.*, 24(4):529–565, December 1999.

[ZSYW10] Fan Zhang, Shuming Shi, Hao Yan, and Ji-Rong Wen. Revisiting globally sorted indexes for efficient document retrieval. In *Proceedings of the*

third ACM International Conference on Web Search and Data Mining (WSDM'10), pages 371–380, 2010.

[ZXL+12] Xujuan Zhou, Yue Xu, Yuefeng Li, Audun Josang, and Clive Cox. The state-of-the-art in personalized recommender systems for social networking. *Artificial Intelligence Review*, 37(2):119–132, 2012.