Efficient Processing of Ranking Queries in Novel Applications

THÈSE N^{O} 4782 (2010)

PRÉSENTÉE LE 13 AOÛT 2010

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES D'INFORMATION RÉPARTIS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Parisa HAGHANI

acceptée sur proposition du jury:

Prof. J.-P. Hubaux, président du jury
Prof. K. Aberer, directeur de thèse
Prof. A. Ailamaki, rapporteur
Prof. Q. Lv, rapporteur
Dr S. Michel, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

To my parents, Ghodsi and Ahmad

Abstract

Ranking queries, which return only a subset of results matching a user query, have been studied extensively in the past decade due to their importance in a wide range of applications. In this thesis, we study ranking queries in novel environments and settings where they have not been considered so far.

With the advancements in sensor technologies, these small devices are today present in all corners of human life. Millions of them are deployed in various places and are sending data on a continuous basis. These sensors which before mainly monitored environmental phenomena or production chains, have now found their way into our daily lives as well; health monitoring being a plausible example of how much we rely on continuous observation of measurements. As the Web technology evolves and facilitates data stream transmissions, sensors do not remain the sole producers of data in form of streams. The Web 2.0 has escalated the production of user-generated content which appear in form of annotated posts in a Weblog (blog), pictures and videos, or small textual snippets reflecting the current activity or status of users and can be regarded as natural items of a temporal stream. A major part of this thesis is devoted to developing novel methods which assist in keeping track of this ever increasing flow of information with continuous monitoring of ranking queries over them, particularly when traditional approaches fail to meet the newly raised requirements.

We consider the ranking problem when the information flow is not synchronized among its sources. This is a recurring situation, since sensors are run by different organizations, measure moving entities, or are simply represented by users which are inherently not synchronizable. Our methods are in particular designed for handling unsynchronized streams, calculating an object's score based on both its currently observed contribution to the registered queries as well as the contribution it might have in future. While this uncertainty in score calculation causes linear growth in the space necessary for providing exact results, we are able to define criteria which allows for evicting unpromising objects as early as possible. We also leverage statistical properties that reflect the correlation between multiple streams to predict the future to provide better bounds for the best possible contribution of an object, consequently limiting the necessary storage dramatically. To achieve this, we make use of small statistical synopses that are periodically refreshed during runtime.

Furthermore, we consider user generated queries in the context of Web 2.0 applications which aim at filtering data streams in forms of textual documents,

based on personal interests. In this case, the dimensionality of the data, the large cardinality of the subscribed queries, as well as the desire for consuming recent information, raise new challenges. We develop new approaches which efficiently filter the information and provide real-time updates to the user subscribed queries. Our methods rely on a novel ordering of user queries in traditional inverted lists which allows the system to effectively prune those queries for which a new piece of information is of no interest.

Finally, we investigate high quality search in user generated content in Web 2.0 applications in form of images or videos. These resources are inherently dispersed all over the globe, therefore can be best managed in a purely distributed peer-to-peer network which eliminates single points of failure. Search in such a huge repository of high dimensional data involves evaluating ranking queries in form of nearest neighbor queries. Therefore, we study ranking queries in high dimensional spaces, where the index of the objects is maintained in a purely distributed fashion. Our solution meets the two major requirements of a viable solution in distributing the index and evaluating ranking queries: the underlying peer-to-peer network remains load balanced, and efficient query evaluation is feasible as similar objects are assigned to nearby peers.

Keywords: ranking queries, top-k, nearest neighbor, data streams, P2P, load shedding, query indexing, filtering, locality sensitive hashing

Zusammenfassung

Anfragen an Informationssysteme, welche nur einen kleinen und sorgfältig ausgewählten Teil an Informationen an die Benutzer wiedergeben, waren ein fester Bestandteil der Forschung in den letzten Jahren. Die sogenannten Top-k-Anfragen, die nur die besten k Objekte an den Benutzer weitergeben, erfordern auf erster Linie ein ausgeklügeltes System an Funktionen, die die Wichtigkeit von Resultaten gemäß Ihrer Informationen wiederspiegeln, als auch effiziente Algorithmen, welche die Informationsverarbeitung auf ein Minimum an Kosten beschränkt. In dieser Arbeit betrachten wir die Informationsverarbeitung von Top-k-Anfragen in Umgebungen und Szenarien die bislang nicht Teil der Forschung waren. Millionen von Sensoren sind in den verschiedensten Bereichen im Einsatz und senden kontinuierlich Daten. Waren Sensoren vormals hauptsächlich in Bereichen wie der Überwachung von Produktionsanlagen und dem Messen von Charakteristiken der Umwelt im Einsatz, so kommen sie mehr und mehr auch im täglichen Leben von Millionen von Menschen zum Einsatz. Die kontinuierliche Überwachung von Gesundheitsdaten ist ein gutes Beispiel dafür. Datenströme werden allerdings nicht nur von Sensoren erzeugt, sondern auch, gerade mit der wachsenden Popularität des Web 2.0, durch von Menschen erzeugten Informationen in Form von Blog-Einträgen, hochgeladenen Fotos und Videos oder auch kurzen Textnachrichten, die den gegenwärtigen Status eines Benutzers wiederspiegeln. Ein Teil dieser Dissertation beschäftigt sich mit der Entwicklung neuer Methoden, die eine effiziente Verarbeitung dieser Datenströme ermöglicht. Zu Beginn betrachten wir bestehende Ansätze zur Top-k-Anfrageverarbeitung in Datenströmen und wie gut diese auf unsere neuen Szenarien anwendbar sind. Wir sehen, dass bestehende Ansätze nur schwer mit Daten umgehen können, die von autonomen Sensoren oder von Benutzern erzeugt werden.

Im ersten Teil dieser Arbeit beschäftigen wir uns folglich mit dem Problem von Top-k-Anfragen im Fall von nicht synchronisierten Datenströmen, wo Informationen über zu observierende Entitäten nicht zeitgleich in allen Strömen zur Verfügung steht. Dieses Verhalten lässt sich in einer Vielzahl von Szenarien beobachten, wie zum Beispiel in Sensornetzwerken, die von verschiedenen Organisationen betrieben werden, Sensoren die verteilt über ein Straßennetz vorbeifahrende Autos beobachten oder ganz einfach Benutzer von Web 2.0 Anwendungen, in denen Daten ohne jegliche Synchronisation erzeugt werden. Unsere Methoden sind für eben diese Szenarien konzipiert. Um Top-k-Anfragen über Datenströme ausführen zu können, wird die Güte eines Objekts basierend auf

den aktuell, in einem Zeitfenster, observierten Eigenschaften berechnet, aber auch unter Berücksichtigung des Beitrags von Eigenschaften, welche in naher Zukunft im System eintreffen. Dieses Vorgehen erzeugt allerdings eine gewisse Unsicherheit in der Berechnung der tatsächlichen Güte, was einen Speicherplatzbedarf linear in der Anzahl der eintreffenden Objekte bedeutet. Die im Rahmen dieser Arbeit entwickelnden Methoden erlauben es dennoch Objekte so früh wie möglich aus dem Speicher zu entfernen. Wir benutzen darüberhinaus statistische Eigenschaften der beteiligten Datenströme, um die Unsicherheit der Qualitätsberechnung zu minimieren.

Im zweiten Teil dieser Arbeit beschäftigen wir uns explizit mit der Verarbeitung von Benutzeranfragen im Kontext von Web 2.0 Anwendungen, welche Ströme von Dokumenten anhand von registrierten Anfragen filtern und nur die besten Resultate an die Anwender weiterleiten. Dies erzeugt aufgrund der hohen Dimensionalität der Daten, der großen Anzahl von registrierten Anfragen zusammen mit der Anforderung aktuelle Informationen zu erhalten, neue Herausforderungen. Die von uns entwickelnden Methoden erlauben eine effiziente Filterung der Informationen und eine Benachrichtigung der Benutzer in Echtzeit. Wir erreichen dies durch eine neuartige Organisation der Benutzeranfragen in traditionellen invertierten Listen, welche dem System erlauben Anfragen die nichts oder nur sehr wenig mit dem eintreffenden Dokument zu tun haben, so bald als möglich zu eliminieren.

Im letzten Teil dieser Dissertation betrachten wir von Benutzern generierte Inhalte in Form von Fotos und Videos. Diese Daten habe zwei Dinge gemeinsam: ihre große Anzahl von Attributen, die bei der Suche in Betracht gezogen werden müssen und ihre große Anzahl an sich, da es immer einfacher wird Fotos und Videos zu erzeugen und diese per Mausklick in das Internet zu übertragen. Wir betrachten Anfragen in diesen hochdimensionalen Räumen, wobei der Suchindex, welcher die Objekte beschreibt, verteilt über viele Rechner gehalten wird. Der von uns entwickelnde Ansatz erlaubt nicht nur eine effiziente Anfrageausführung, sondern auch eine gleichmäßige Verteilung des Suchindexes über die beteiligten Rechner.

Stichworte: Top-k-Anfragen, Nächste-Nachbar-Suche, Datenströme, P2P, Lastabschaltung, Anfrageindexierung, Filter, Locality Sensitive Hashing

Acknowledgements

I would like to thank my supervisor, Karl Aberer for giving me the opportunity to pursue my Ph.D. under his supervision. His trust and patience gave me the freedom to explore different problems, while he guided me through the way with invaluable insight.

I also thank the members of my committee, Anastasia Ailamaki, Christine Lv, and Sebastian Michel, for the time they spent in reviewing this thesis and their constructive feedback on the manuscript. I thank Jean-Pierre Hubaux for agreeing to be the president of my committee.

I am very grateful to my colleagues from the LSIR lab for all the fruitful discussions and friendly moments. I specially thank Chantal for her endless logistical support. Another very big thank-you goes to Sebastian, from whom I learned a great deal, in summary, *how to be a golden PhD student*, and to Adriana, for being the best possible officemate. I made amazing friends here and I am grateful to them for all the unforgettable moments we had together: Abbas, Atefeh, Audrius, Dan, Fabius, Fereshteh, Gleb, Ingmar, Irinka, Iuli, Ivana(s), Julien, Kasia, Laleh, Maciek, Mahdi, Marcin, Marta, Maxim, Michal, Nevena, Oana, Olga, Phil, Razvan, Roman, Sara, Sarunas, Simas, Shahram, Shokoufeh, Soheil, Vahid, Valka, Wojtek, Zahra, Zoltan, and many many others.

I wish to express gratitude to my parents for their truly unconditional love and support, which despite being far away I always felt here in Switzerland. Thanks to Sasan and Laleh for their love and advise, and to Sina for our cheerful everyday chats. Finally, thanks to Saman, my *someone*, for his encouragements, love and support.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1

Introduction

Ranking queries [IBS08, BBK01] constitute an important technique for focalizing attention to the most essential results of a query. In order to deal with massive quantities of data, such as in multimedia search, Web search, and distributed systems, data from the underlying database are scored according to an application-dependant scoring function and then ranked based on their scores. The ranking query returns the tuples with the highest rank among others. A ranking query can therefore involve evaluating a predicate on the database, executing joins to combine tuples from different tables, or performing grouping based on similar attributes and finally scoring and ranking the output. This class of queries have enormous applications in a variety of problems and are nearly ubiquitous in all data related operations. Consider the following example:

Example 1 In image databases (e.g., [BEF+09]), hundreds of millions of images are processed and stored inside a database, producing massive amounts of data. Content based image retrieval in such databases is traditionally carried out using high dimensional indexes built over single image features. Given a similarity function which aggregates different features' similarity scores into an overall score, the relevance of each image in the database, to a query image, can be measured. Suppose the user is interested in the top-10 images which are most similar to a given image based on color and texture features. Let the query image be represented by q and let i represent an image in the database. The function $sim(q, i) = colorSim(q, i) + textSim(q, i)$ could be regarded as an aggregation function which combines the similarity scores based on color and texture features. The ranking query returns 10 images with the highest overall similarity score to the query image.

The efficiency and effectiveness of search in the above example depends highly on the ranking algorithm. Similar applications exist in the context of Web search (e.g., meta-search), information retrieval, and data mining. Processing ranking queries connects to many traditional database research areas, such as query optimization, query languages, and indexing methods, where it

Characteristic	Impact
(C1) Dynamic data	push-based data access continuous queries
(C2) Massive data volumes	impossible to store all data locally impossible to check all local data
(C3) Communication	data unavailability data incompleteness

Table 1.1: Major characteristics of novel environments and their impacts

has been extensively studied previously.

Nevertheless, the sweeping changes brought about by the digital revolution in the computing and communication technology in the last decade, has given rise to the need for different kinds of data management systems [Sel08]. Traditional databases which were built in controlled environments under certain assumptions, and stored almost static data, are ill-equipped for some of today's applications. Consequently, new measures have to be taken in to account to assure high performance and availability in all data-centric operations, including ranking queries. As an instance of a ranking query application in a non-conventional database setting let us consider the following example:

Example 2 Digital photography and its widespread use have changed the photo industry dramatically. With the rapid availability of high tech cheap digital cameras, photo shooting and sharing has become a popular activity. Today, several Web 2.0 portals devoted to photo sharing exist, in which, space is dedicated to users for storing their photos and making it possible for others to view them. Uploading and commenting on photos are almost effortless with the provided user friendly interfaces. Many users subscribe to such a system to follow images of their favorite events or people. Keeping a user updated, usually involves continuously executing ranking queries over the stream of newly uploaded photos to retrieve the top most similar images to the user's pivot images.

The image database of the portal described above bears fundamental differences to the traditional database of Example 1. While queries in a traditional database operate on persistently stored tables, queries of Example 2 act upon a real-time stream of data values. Human-generated data, like in the example above, constitutes only a very small portion of the multitude of real-time information produced nowadays, a credible example being the reported measurements of sensor networks deployed in various places. The spontaneous nature of these information sources has opened a trend towards push-based data access which is in contrast with the often pull-based accesses in traditional databases. In traditional databases queries are considered as *one-shot* and terminate as soon as they produces the results while given dynamic data, queries are run *continuously*.

Massively growing data volumes, as another consequence of the digital revolution, pose new challenges in storage and management of data. In many data-centric applications, it is not possible anymore to store all data on a sin-

gle machine. Distributed databases and peer-to-peer systems have emerged as remedies to this problem. Distributed databases are suited for well-controlled environments, while peer-to-peer overlays are usually formed on a self-organizing network of computers, which potentially offer more flexibility and higher capacity. Measures which define and affect system performance in these systems are different from traditional centralized databases: network delay dominates local CPU computation time, and load balance is a new metric affecting availability. Furthermore, even if all data could be stored on a single machine, in applications requiring real-time responses, often all data which could contribute to the final query results can not be checked. As a result of the above mentioned issues, data partitioning and load shedding algorithms are required.

Distributed sources of information along with distributed storage and computing devices highlight the impact of communication in today's new applications. Smart phones, highly mobile tablet and laptop devices now offer powerful platforms for acquiring and delivery of new services whose availability depend on reliable communication. In the data stream model, sources of information may deliver out of order data to the processing unit, causing data incompleteness. On the other hand, in distributed systems unavailability is usually due to some storage or computing nodes becoming inaccessible. Data incompleteness and unavailability is therefore another issue requiring special care in novel data management systems.

The motivations behind this thesis are rooted in the limitations of traditional database systems in evaluating ranking queries in novel data management systems, more specifically in *data streams* and *peer-to-peer networks*. Table 1.1 summarizes some major characteristics of these novel systems and their impact which drives them different from traditional databases.

1.1 Challenges and Contributions

We have identified settings in new environments in which ranking queries are of great importance, but traditional techniques fail to ensure high performance and availability. This failure is due to one or a combination of the characteristics listed in Table 1.1. Measures of interest and sources of complication are different in each setting, so we have developed several novel techniques to enable efficient processing of ranking queries in each. Table 1.2 provides a short summary of the main sources of complications we have considered, showing which measure of interest they influence and our proposed solution for each. In the following we discuss in more details the specific problems which have been considered in this thesis and our contributions towards each.

Top-k Queries Over Data Streams

In general, the data stream model applies to any dynamic system which requires to be monitored continuously. Communication and network monitoring [GKMS01, SH98], environmental monitoring [MSL+09], and financial data anal-

characteristic	influencing measure	solution
C2+C3	memory	semantic load shedding
C1+C2	CPU time memory	query indexing, data filtering result maintanance
C2+C3	network delay	locality preserving mappings

Table 1.2: Characteristics which are sources of complications, their influencing measure and our proposed solution

ysis [CDTW00] are instances of such systems. Ranking queries naturally arise in such settings and have been previously studied [MBP06, PZA08] with regard to different criteria, such as space considerations or performance in terms of CPU time for evaluating the queries. However, all existing approaches assume that exact score calculations are possible for objects as they arrive in the system. Nevertheless, in many streaming scenarios this is infeasible. A prior join over multiple non-synchronized streams may be necessary before the score of an object can be calculated. Our first main contribution in this thesis is the introduction of a *semantic load shedding* algorithm under such a setting, which aims at limiting the necessary space, while maintaining the precision of the top-k evaluation. More specifically:

- We consider continuous monitoring of aggregation queries over multiple non-synchronized data streams in a sliding window model which has not been, to our knowledge, considered previously.
- We define the notion of *dominance* under incomplete information such that the two necessary properties of it, namely *persistency* and *transitivity* are attained. With this, we enable retaining only those objects which are necessary for providing exact results to the registered top-k query.
- We theoretically show that the necessary memory usage increases from previously logarithmic (in window size) to linear, compared to the case where data is received completely and exact score calculations are possible.
- Leveraging statistics collected from the data streams and their correlations, we propose an approximate algorithm which allows for early-drops of objects with little loss in the accuracy of the returned results.

While the data stream model has been successfully used in modeling applications involving data operations over communication and networking systems, financial feeds, or sensor data, Web 2.0 related problems, mainly due to their recency, have been left out. With the advent of Web 2.0, yesterday's end users are now content generators themselves and actively contribute to the Web. Each user action, for example uploading a picture, tagging a video, or commenting on a blog, can be interpreted as an event in a corresponding stream. Given the immense volume of this data and its vast diversity, there is a vital need

for effective filtering methods which allow users to efficiently follow personally interesting information and stay tuned. Ranking queries, which correspond to user defined profiles, act as filters which should be continuously updated, in order to fulfill the universal tendency for newly published information. We envision a scalable filtering system which relies on subscription of hundreds of thousands of profiles as top-k queries, and real-time monitoring of each. Our contributions to this problem include the following:

- We design an efficient profile filtering algorithm, which refrains from processing all profiles with every incoming document. We show that our method is exact and all profiles which a new incoming document has a chance of being their top-k result are identified.
- We use a skyline based method for result maintenance, but in order to avoid inserting *all* incoming documents to the result set, which is the case for in-order streams, we restrict the insertion criteria such that the size of the maintained result set remains small. We derive the necessary conditions to insure exact results.

Nearest Neighbor Queries Over P2P Networks

A P2P network is an *overlay network*, built on top of a native or physical topology, which provides a platform for distributing various tasks among numerous linked devices. Top-k query processing has been previously studied in the P2P setting [MTW05a] where the inverted lists are distributed among peers. Efficient algorithms which guarantee exact results for a top-k query in a limited number of phases [CW04] exist. On the other hand, similarity search over high dimensional data, as another kind of a ranking query studied extensively in centralized settings previously [BGRS99, GIM99, YOTJ01, BBK98, DIIM04], has proven to be more difficult. Existing approaches to the similarity search problem in high dimensional data either focus on centralized settings, as cited above, rely on preprocessing data centrally, assume data ownership by peers in a hierarchical P2P setting or fail at providing both high quality search results and a fair load balance in the network [FGZ05, SEAA04, DVKV07]. In this thesis we address this problem and provide solutions which satisfy the necessary conditions for efficient similarity search in a P2P setting. Our contributions include the following:

- We discuss the difficulties of distributing existing Locality Sensitive Hashing (LSH) [GIM99] schemes and derive requirements to distribute them in a way that assures fair load balance and efficient and accurate similarity search processing.
- We present two novel mapping schemes which satisfy the mentioned requirements.

- We present a top-k like algorithm which leverages the locality preserving property of our mapping schemes to efficiently process distributed kNN queries.
- We show, relying on our mapping schemes, how otherwise-difficult-to-process range queries can be efficiently processed in our setting by presenting a novel sampling-based method which utilizes our estimated range of peers necessary to contact.

1.2 Publications

This thesis is based on the following publications.

In the area of continuous top-k processing over data streams in a sliding window model:

In [HMA09b], we have addressed the problem of evaluating top-k queries over multiple non-synchronized data streams. We have developed an exact algorithm which retains only the necessary objects to provide exact results and drops the rest in order to save on space. We have shown that even only retaining necessary objects incurs linear space in size of the sliding window. Based on the observation that the final scores of not completely seen objects could be better estimated based on the inter stream correlations, we have developed an approximate algorithm which rigorously decreases the space overhead, while maintaining high accuracy in results. This paper is described in Chapter 4.

- Parisa Haghani, Sebastian Michel, Karl Aberer: Evaluating top-k queries over incomplete data streams. 18th ACM Conference on Information and Knowledge Management (CIKM2009), HongKong, China, November 2-6 2009.

In [HMA10], we consider the problem of continuous evaluation of large number of top-k queries over a stream of textual data under a sliding window model. The motivation behind this work is the design of an information filtering system which can serve hundreds of thousands of users and keep them updated on newly published data on the web based on their personalized profiles. In order to avoid comparing each incoming object to all registered queries, we index the queries in a way that allows for early stopping. We further introduce a relaxed version of our previously proposed sorted list indexes which maintains the accuracy of results, while incurring much less update cost. This paper is described in Chapter 5.

- Parisa Haghani, Sebastian Michel, Karl Aberer: The Gist of Everything New: Personalized Top-k Processing over Web 2.0 Streams. to appear in

CIKM2010, Toronto, Canada, October 26-30 2010.

In the area of k nearest neighbor queries over P2P networks:

In [HMA09a, HMCMA08] we have considered the problem of k nearest neighbor search over high dimensional data over a P2P network. To escape from the curse of dimensionality, we proposed to use the well known Locality Sensitive Hashing [GIM99] scheme and distribute it over the peers of a network. Our solution satisfies the two necessary requirements for efficiency and robustness: similar objects are likely to be placed on the same peer or neighboring peers, which assures efficient k nearest neighbor search, furthermore data is assigned to peers in such a way that ensures a fair load balance over the network. These two papers are described in Chapter 6.

- Parisa Haghani, Sebastian Michel, Karl Aberer: Distributed Similarity Search in High Dimensions Using Locality Sensitive Hashing. 12th International Conference on Extending Database Technology (EDBT 2009), Saint-Petersburg, Russia, March 23-26 2009.

- Parisa Haghani, Sebastian Michel, Philippe Cudre-Mauroux, Karl Aberer: LSH At Large - Distributed KNN Search in High Dimensions. 11th International Workshop on Web and Databases (WebDB 2008), Vancouver, Canada, June 13, 2008.

1.3 Outline

This thesis is organized as follows. In Chapter 2, we introduce the fundamental concepts used throughout this thesis. We start by giving an overview of the data stream model and briefly discussing important problems in this domain. Next, we introduce the peer-to-peer computational model and give a summary of such existing networks. Then we move to ranking queries where we divide this class of queries into top-k and k nearest neighbor queries, and review some classical techniques in each. In Chapter 3, we give an overview of state of the art techniques, related to our specific problems. In Chapter 4, we discuss continuous top-k processing over multiple non-synchronized data streams in a sliding window model. We show that the space consumption necessary for providing exact top-k results grows linearly in the size of the sliding window. We introduce a probabilistic algorithm which further limits the space needed while maintaining the precision. In Chapter 5, we consider another application which involves monitoring large number of continuous queries over a stream of text data. The main concern in this setting is performance and indexes are designed such that each incoming document does not need to be compared with all registered top-k queries. We introduce the *POL-filter* which is a relaxation

of our completely sorted index previously introduced as the first solution to the problem at hand. In Chapter 6, we consider one-shot k nearest neighbor queries over high dimensional data in a peer-to-peer setting. Finally in Chapter 7, we conclude the thesis and give directions for future work.

Chapter 2

Background

In this Chapter we will first give an overview of the *data stream* and *peer-to-peer* computational models which are our underlying environments. We later discuss top-k and kNN queries which are the type of queries we consider in this thesis.

2.1 Emergent Novel Environments

Traditional DataBase Management Systems (DBMS) have served the main information technological needs of our community for more than three decades. Utilizing a DBMS, it is easy to store large amounts of data and manipulate it as desired, by performing various queries on the data. In recent years however, new classes of applications have emerged which call for functionalities beyond the primitives of a traditional DBMS.

One prominent and well addressed member of these unconventional information management systems are *Data Stream Management Systems*. While conventional DBM systems are designed to handle static data and to process *one-time* queries over such data, many of today's applications deal with data which is generated in an unbounded fashion and require continuous monitoring. The *Data Stream* model has emerged as an alternative for dealing with such data and providing solutions to specific challenges raised in such an environment.

On the other hand, with the growing number of interconnected computers which is mainly due to the increasing popularity of the Internet, the peer-to-peer (P2P) computational model has emerged as a platform for various tasks spread over numerous devices. As one of the most prominent examples, the SETI@home approach leverages the computing power of thousands of standard computers when they are idle – running a screen saver. In general, P2P is often a synonym for file-sharing applications (e.g., Napster[1], Gnutella [2]), where users exchange content without any central control instance. Another phenomena which gives rise to such a model is the scatteration of content generators in

[1] http://en.wikipedia.org/wiki/Napster
[2] http://en.wikipedia.org/wiki/Gnutella

geographically distant locations. While traditional DBMS are suited for well-controlled and known environments, they lack the required characteristics of managing data in a highly dynamic and potentially diverse setting.

In the following we will give an overview of these two emerging non-conventional environments which are the underlying environments we consider in the future chapters.

2.1.1 Data Streams

Research on data streams has gained a lot of interest in the past few years [BBD+02, Mut05, CKT08, MBP06, KOT04, BOPY07]. The foundations and applications of this research topic are found in many domains, including databases, data mining, algorithms, networking, theory, and statistics. Many data and query characteristics of modern applications are best captured by this computational model, where data streams in continuously at high rates and each tuple has the chance of being observed only once, unless explicitly stored for later accesses. Some applications are for instance:

Communication and network monitoring: Internet traffic analysis [GKMS01, SH98] has been a primary application of data streams. An Internet Service provider monitors the traffic at various points in the network to provide better resource allocations or thwart possible security attacks. According to the specific application, incoming data can have different granulation and schema, for example the data can consist of detailed logs on a per flow granularity.

Environmental monitoring: Deployment of low-cost sensors in hard-to-reach but critical places has given rise to various environmental monitoring applications [Swi, MSL+09]. These sensors usually report on different aspects of the environment such as temperature, luminosity and humidity. Each observation is tagged with a time-stamp which indicates the time of observation and is sent to a central processing unit for further analysis. The online analysis of data usually includes searching for indications of natural hazards and distributing them in a timely manner. The data is archived for usage in forecasting or validating existing models.

Web 2.0 Streams: With the advent of Web 2.0, yesterday's end users are now content generators themselves and actively contribute to the Web. Each user action, for example uploading a picture, tagging a video, or commenting on a blog, can be interpreted as an event in a corresponding stream. Given the immense volume of this data and its vast diversity, there is a vital need for effective filtering methods which allow users to efficiently follow personally interesting information and stay tuned.

Internet Advertising: Contrary to TV or Radio broadcast-based advertising, Internet advertisements target the appropriate customer on the fly.

Therefore online decision making on which advertisements are suitable for a user given his/her click stream structure is very important [MAA05]. On the other hand, given the specifications of Internet advertisement, click fraud can jeopardize this industry. Effective measures against this kind of fraud call for efficient stream analysis which enable attack detection [MAAZ07].

Financial data analysis: In today's financial market, fast decision making on multitudes of financial feeds is necessary for successful trading actions. Large number of users may subscribe their queries to a central unit [CDTW00] which needs statistical monitoring of numerous financial data streams [ZS02] to accurately provide results to the queries.

The main differences of the streaming model with traditional database models include the following:

- **Unbounded Data:** In traditional DBM systems, the data is stored on disk usually in form of a relation and additional structures such as indices on certain attributes are used to speed up the process of accessing specific parts of the data. This data can be considered as static as changes to it occur with low frequency and the main characteristics of the data distribution remain unchanged over a long period of time. In the streaming model however, the data arrives continuously and the data stream manager does not have any control on the rate or amount of received data. As a result the usual database techniques for storing and indexing the data may not be efficient in this setting. Due to the often immense rate of incoming data, tailored approaches for memory management are required.

- **Continuous Queries:** The queries in traditional DBMS are usually regarded as *one-time* queries. They are run once over the data and terminate with returning the results. On the other hand we face another type of queries in the streaming model, in the sense that they are *continuous*. In streaming applications, queries are posed on one or several unbounded streams and require updates in the results as new data arrives or old data times out according to an expiration model.

Algorithms and Systems

Early works on stream processing mostly consider one-pass algorithms in limited space over the whole stream where all tuples are considered valid at all times. These algorithms can be divided into two main groups: *sampling* and *sketches*. In *sampling* based methods, a sample of the whole stream is maintained to provide fast approximate answers to queries such as distinct value estimation [Gib01] or keeping statistical summaries of the stream such as histograms [GMP97]. FM sketches [FM85] which are among hashing sketches are

used for distinct item counting over streams. AMS [AGMS02] sketches, based on linear projections, are used to estimate the self-join size of one stream or join-size of several streams. Reporting on *quantiles* or *heavy hitters* in streams is another important problem studied in the literature: Solutions often apply techniques such as the mentioned AMS sketches, sampling, or more recently group testing, see for example [CCFC04, CM03] and the references within. The exponential histogram technique [DGIM02] and deterministic waves [GT02] maintain stream statistics such as bit counting or sum over a sliding window in streams. In this model only tuples within a sliding window are considered valid. Different variations of the sliding window model are the *counting* and *time-based* models, where in the first, statistics over a certain number of most recent tuples is of interest and in the second these statistics are maintained for all tuples which have arrived within a time frame. In a more general model, Cormode et al. [TXB06, CKT08] consider time decaying aggregates in out-of-order streams. There has also been some work on uncertain data streams where the tuples arriving are associated with existential probability. [CG07] devises sketches for estimating number of distinct elements and size of joins over this kind of streams, while [AY08] proposes a framework for clustering them.

In recent years *Data Stream Management Systems* (DSMS) have been proposed which address new challenges in data management and query processing that arise in the context of stream processing. STREAM [ABB+03] is a DSMS which supports a large class of declarative continuous queries over data streams. CQL (Continuous Query Language), an expressive SQL-based declarative language is supported by STREAM for registering continuous queries [ABW03]. Auorora [ACc+03] is another model and architecture for data stream management. Unlike STREAM which supports CQL, Aurora assumes more direct, workflow-style specification of queries. Aurora drives its resource management decisions, such as scheduling, storage management, and load shedding, based on various QoS specifications.

2.1.2 Peer-to-Peer Networks

The peer-to-peer (P2P) approach aims at utilizing the resources of numerous linked computers with diverse processing or storage specifications in a distributed fashion. While most of today's popular Web applications such as search engines, sharing portals and email services are deployed over a large number of servers to handle high workloads, they still rely on a centralized station. Any centralized point is prone to error and its failure can cause a whole network of services to go non-functional. The P2P paradigm however eliminates the central point of failure and empowers every involved peer with a degree of autonomy, i.e., peers act as both *servers* and *clients*. This approach offers significant advantages in terms of scalability, efficiency and resilience to failure and dynamics. According to a recent study, P2P applications constitute the largest bandwidth consumers of the Internet. File sharing applications (e.g., BitTorrent [3]) and

[3] www.bittorent.com

P2P telephony (e.g., Skype [4]) are examples of widely used applications of the P2P approach.

A P2P network is an *overlay network*, built on top of a native or physical topology, for example, the Internet. Nodes in a P2P network are called *peers* and in general act as both *clients* and *servers*. Any node in the network can request a resource which is held by another peer in the network. Since the number of peers in a P2P network is usually very high and they can be located in geographically far positions from one another, efficient means of finding the peer which is holding a required resource is key in the applicability of a P2P network. Various indexing methods (e.g., a map from the resource to the peer holding the resource) have been proposed for this problem. *Central directory-based* P2P systems (e.g., Napster) use a central index to keep the index, while resources are distributed among peers. This solution suffers from the *single point of failure* problem and is not scalable. On the other hand, decentralized P2P systems [SMK+01] use a distributed index, i.e., the index is also distributed among the peers and if a peer fails only a fraction of index may be unaccessible. Between these two indexing methods there also exists a *hybrid* alternative. In a hybrid P2P network [Gnub], the network is divided to several subsets and each subnetwork has a a super-peer which plays the role of a central index holder for that subset of peers.

Several measures are used to assess the suitability, robustness, and efficiency of a P2P network. The communication cost is an important measure which is usually defined as the number of network messages exchanged in the network until a resource is found. Another factor is the query response time, i.e., the delay between a request and its response. The load balance in the network is also an important factor which affect the robustness of the network and its goal in avoiding single points of failure.

P2P networks can also be classified as *structured* [SMK+01] or *unstructured* [Gnua] based on the level of knowledge each peer maintains regarding the other peers and the network in general, as well as the structural properties of the network. Unstructured P2P networks do not impose a specific structure on the peers forming the network and the network is organized according to an arbitrary, usually random, topology. Searching in unstructured P2P networks incurs high costs but the network is more robust as peers are keeping many links to other peers. Searching methods in unstructured P2P networks are usually based on random walks [CRB+03] or constraint broadcasts. In constraint broadcasting, a request message is broadcasted from a peer to all its neighbors and all neighbors repeat the same until the message expires. A Time-To-Live (TTL) counter specifies the liveness of a message which can be time-based or depend on the number of peers which have forwarded this message so far. Clearly this method can flood the network with large number of messages. In order to avoid this overhead, the search method based on random walks, transmits the request message to one or more *randomly* selected neighbors until the resource is found or the message expires. This method on the other hand can cause higher delay

[4] www.skype.com

times.

In structured P2P networks a globally consistent protocol is employed to ensure that any node can efficiently route a search to some peer that has the desired resource. While such a protocol decreases the search cost significantly compared to the unstructured methods, it necessitates a more structured pattern of overlay links. For example peers joining and leaving the network trigger actions by other nodes which should update their entries regarding the new peers or the left ones. The distributed hash-table (DHT) is by far the most common type of structured P2P network. In a DHT, a hashing method is used to assign ownership of a resource to a particular peers.

In the following Section we describe the Distributed Hash Table structure and then P-Grid [Abe01] and Chord [SMK+01], as examples of such a structure are discussed.

Distributed Hash Table (DHT)

A Distributed Hash Table (DHT) is a distributed structure which provides a lookup service similar to a traditional hash-table. In the P2P context, each resource is mapped to a *key* using a given hash function and each peer also carries a particular ID, which is usually in the same key-ID space. The routing layer of the network takes care of determining for a given key the peer that is, so-called, responsible for the key, e.g., its ID is numerically closest to the key. With this routing, a storage of $\langle key, value \rangle$ pairs can be build by routing the pairs using their keys to the responsible peer that then store these pairs. Similarly, to retrieve a *value* for a given key, the key is used to route the *get* request to the responsible peer which will in return send back the matching *value*. Each peer in the DHT is responsible for a fraction of the overall key-space.

In order to facilitate efficient search, each peer maintains a *routing table* to forward the search messages which can not be answered locally to its neighbors. Routing tables are always constructed such that the completeness property is guaranteed, i.e., they cover the whole key-space and the search for a resource can be initiated by any peer in the network. The routing table of each peer contains peer identifiers of those peers which are positioned in an exponentially increasing distance from this peer in the key space. With this technique, a targeted peer is found in the network after $O(logN)$ peers have been contacted, where N is the number of peers in the network. All variations of DHTs are essentially based on this technique and vary in issues such as variable key space partitioning versus fixed partitioning, the topology of the key space (interval, ring, torus, etc.), how the routing information is kept updated with regard to redundant entries and dealing with churn and network dynamics.

P-Grid [Abe01], Chord [SMK+01], CAN [FGZ05], and Pastry [RD01] are popular implementations of the DHT structure. In the following we will describe P-Grid and Chord.

P-Grid

P-Grid [Abe01] is a tree-based structured P2P network. Peers and keys are mapped to the same binary key-space represented by a tree, and each peer is responsible for a partition of the overall tree. Note that the tree is just an abstraction and does not represent the actual connectivity of the peers. A peer's position in the tree is determined by a binary bit string, called the path, which also represents the subset of the keys that this peer is responsible for. In order to have efficient key lookups, each peer stores pointers to other peers in the following way: for each bit in its path, it stores references to at least one other peer that is responsible for the other side of the binary tree at that level. At lookup time, if a request can not be answered locally, the peer forwards the request to a peer that has the *closest* path to the desired key, i.e., it has the largest common prefix with the key. Since the peer paths are not determined a priori and can be changed dynamically through negotiations with other peers, P-Grid is robust with regard to skewed data distributions. It has been shown that for a sufficiently random selection of links to other peers, the search cost in terms of messages exchanged among peers remains logarithmic in number of peers in the network.

Chord

In Chord [SMK+01], all peers and keys are mapped to the same one-dimensional cyclic key-space. The key-space forms a ring, such that the first ID (i.e., 0) follows the biggest ID. For simpler representation in the following we do not show the hash function but treat peer IDs as if the hash function has been applied to them. Let p_i represent the ID (after hashing) of the i-th peer and similarly let k_i represent the i-th key. The keys are allocated to the peers in the following way: The peer whose ID most closely follows a key k is responsible for it and is called the *successor* of k. Therefore each peer is responsible for all keys with identifiers between the ID of its predecessor peer and its own ID. Figure 2.1 shows an illustration of a Chord ring. Eight peers have been placed on the ring and five resource allocations are shown as examples. Each peer knows its immediate neighbor and in case of a lookup it contacts its immediate neighbor if it cannot answer the search locally. For example a lookup for k_{56} at p_2 is forwarded to p_{15} where it is forwarded again until it reaches p_{64}. The cost of this greedy search, where the available routing information per peer is $O(1)$, is linear in the number of peers. However, as previously mentioned peers also hold routing information to speed up search. The routing information in Chord are saved in so-called *finger tables*. The m-th entry in the finger table of a node p_i points to a peer p_j where p_j is the smallest ID which proceeds p_i by at least 2^m on the Chord ring. The finger table therefore contains information regarding a limited number of other nodes. However, as each peer has finger entries at power of two intervals around the Chord ring, the distance to the desired peer is at least halved with each forward. An example of this is illustrated in Figure 2.2 where the finger tables are shown for p_2 and p_{35}. Since the search distance is

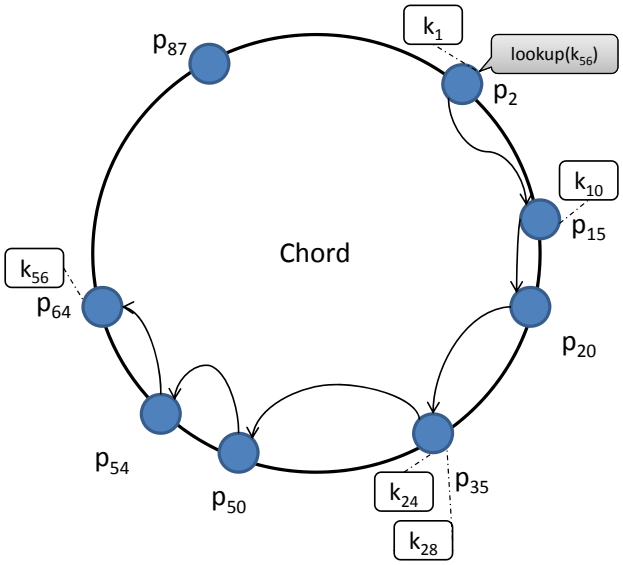


Figure 2.1: The Chord ring

halved with each forward, every search can be answered by contacting at most $O(log(N))$ other peers.

2.2 Ranking Queries

In many applications, the number of returned results are limited by calculating an aggregate over a subset of attributes and returning values above a given threshold, or, returning only the highest values. For example, assume that the underlying data resides in a table with the following schema: `Events(item,attribute1, attribute2,...,attributem)`. Given a threshold T, the first kind of query which limits the results by T, is called an *iceberg query* [FSGM+98] and has the following form:

```
SELECT item, Aggr(attribute1,...,attributem)
FROM Events
WHERE Aggr(attribute1,...,attributem)>T
```

This kind of queries arise in many applications including data warehousing, market basket analysis in data mining, and copy detection. The other kind of queries which bound the number of returned results are the so-called *top-k aggregation* queries and can be expressing in the following form:

```
SELECT item, Aggr(attribute1,...,attributem)
FROM Events
ORDER BY Aggr(attribute1,...,attributem) DESC
LIMIT k
```

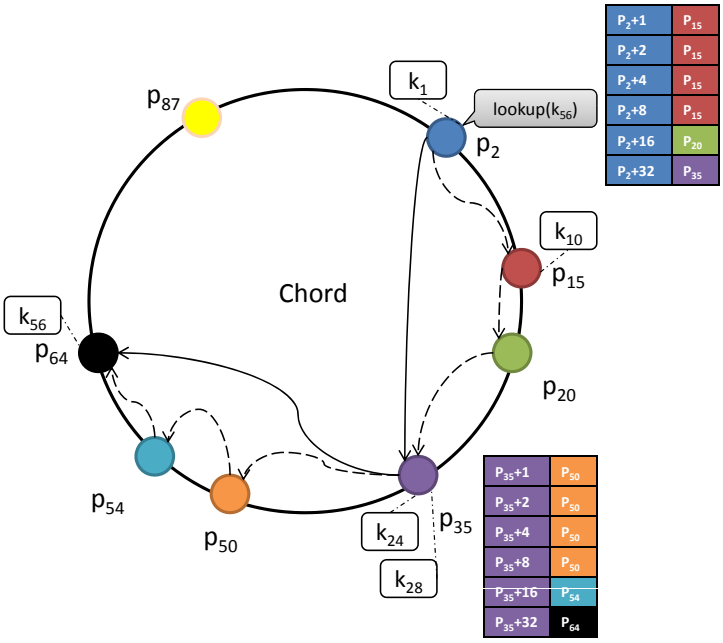
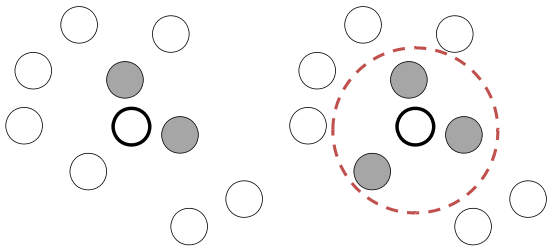


Figure 2.2: Finger tables and logarithmic lookup.

Essentially, an iceberg query can be evaluated using a suitable top-k aggregation query and vice versa. Given the threshold parameter T in an iceberg query, in order to transform the query to a top-k aggregation query, the appropriate k should be chosen, such that all the returned results have aggregation values higher than T and all those which satisfy this criteria are returned. If the distribution of the aggregate value is known, a primary value for k can be estimated and tuned by repeatedly evaluating the corresponding top-k query and viewing the results.

Another kind of queries which also involves ranking database items based on an aggregate of their attributes are the so-called *query by example* or *nearest neighbor* (NN) queries. Assume items in a database are characterized by a collection of their relative features (attributes) and presented as points in a multi-dimensional space. Given a similarity (distance) measure between these points and a query in form of the points in this space, the goal is to find the most similar (smallest distance) items to the query. This problem is of major importance to a variety of application, such as information retrieval, multimedia search, and pattern recognition. Query by example also comes in two flavors, first are the k *nearest neighbor* (kNN) queries which return the k closest objects to the query point, and second are *range* queries, which return all objects in a distance r of the query point. Figure 2.3 (a) illustrates a kNN search, where the query point is shown in bold and $k = 2$. Figure 2.3 (b) shows an example of a range query.



(a) (b)

Figure 2.3: (a)kNN search with k=2 (b) range query

Similar to top-k and iceberg queries, a kNN query can be evaluated using a suitable range query and vice versa. In all the four discussed types of queries, an effective implementation avoids scanning the whole database when answering a query. For simplicity, in the following we will mention kNN and top-k queries, keeping in mind that range and iceberg queries can be evaluated similarly.

Although kNN and top-k queries bear many similarities, they have significant differences which make techniques suitable for one, inapplicable for the other. The first difference is that the aggregation function considered in a top-k query is usually applied on a *subset* of all attributes, with usually a much smaller cardinality compared to the whole set of attributes. In a kNN queries, the query point is in the form of a point and the distance function usually considers *all* attributes. Furthermore, structures maintained for effectively performing a top-k query, usually consider the class of monotonic aggregation functions. In kNN query however, the aggregation function changes each time the query point changes and is not monotonic.

2.2.1 Top-k Aggregation Queries

Top-k query processing constitutes an important class of database queries and has received great attention in recent years. Among the most successful approaches in terms of efficiency is the family of threshold algorithms (TA) [FLN03, NR99, GBK00]. The TA algorithms are applicable for monotonic aggregation functions.

Definition Monotonicity: Given a function $f(x_1, ..., x_m)$, we say f is monotone if $f(x_1, ..., x_m) \leq f(x'_1, ..., x'_m)$ whenever $x_i \leq x'_i$ for all i.

Most of the aggregation functions which are used in practice are monotone, therefore the TA algorithms are applicable in many real world scenarios. The TA algorithm is based on the existence of per attribute sorted lists. Variations of the TA algorithm depend on whether random accesses to the lists are allowed or prohibited. Several extensions to the TA algorithm exist in the literature: approximate results are discussed in [TWS04] with probabilistic guarantees. Assuming availability of statistics on the lists, [BMS+06] leverages this information

to optimize the performance of the TA algorithm. Utilizing results of previous queries or available combination lists are considered in [DGKT06, KPSV09]. In the following subsection we describe the basic TA algorithm assuming random accesses are allowed.

Threshold Algorithm

Given the schema `EVENT(item,`x_1`,....,`x_m`)` where `item` is a primary key, a sorted list l_i is constructed for x_i for all i. The entries of l_i have the following format: $\langle item, x_i \rangle$ and are sorted in decreasing order based on values of x_i. We assume random accesses are allowed: given the primary key *item* of a tuple, the value of its i-th attribute can be known by a random access to l_i. Given a monotonic aggregation function $f(x_1, ..., x_m)$, the following steps are performed in the TA algorithm:

1. Do sorted accesses in parallel on each of the lists l_i. For each observed item e calculate its aggregation value by performing random accesses to the other lists to retrieve its missing attribute values. Calculate the aggregation function f over e. If $f(e)$ is currently among the scores of the k items with the highest scores, insert e to the set of top k items and discard the $(k+1)$-th item.

2. Let $\underline{x_i}$ denote the value of the last observed item in l_i. Calculate τ as the aggregated value of all last observed values in all lists: $\tau = f(\underline{x_1}, ..., \underline{x_m})$.

3. Stop the above procedure if τ is smaller than the aggregated value of the ranked k item.

The stopping condition in the TA algorithm guarantees that the returned top-k items are the *true* top-k items: as if the algorithm would scan all the items and return the top-k results. It is easy to see that since the lists are sorted in decreasing order, the value τ calculated at each round is the upper bound for aggregated values of items not seen so far. Therefore, if τ is smaller than the ranked k aggregated value, no unseen item can have a better aggregated value.

2.2.2 Nearest Neighbor Queries

Query by example, *similarity search*, or *nearest neighbor* search has been a very important problem in a variety of applications such as multimedia search, pattern recognition, and machine learning. In this problem, the objects of interest are represented by their features in a multi-dimensional space and the query is of form of these objects. Given a similarity measure between the objects, the goal is to retrieve the closest objects of the collection to the query point. In k nearest neighbor (kNN) query the k closest objects are returned. While efficient solutions for this problem exist in low dimensional data (2-4) [Gut84, Ben90], the problem grows much more difficult with increasing dimensions.

kNN Queries in High Dimensional Data

With increasing dimensionality, a broad variety of mathematical effects surface up. These can severely affect the performance of the index structures built for kNN queries and are coined as *curse of dimensionality* in the database community. Many properties which appear in 2 or 3 dimensional space, where people can visualize the space therefore tend to use it as an analogy of higher dimensional space, do not hold in high enough dimensions. For better understanding, let us consider the following example which was first presented in [BBK01]. Consider a cubic-shaped d-dimensional data space of extension $[0,1]^d$. Let c denote the center point of the the data space defined as $(0.5, ..., 0.5)$. Consider the following lemma: *Every d-dimensional sphere touching (or intersecting) the* $(d-1)$*-dimensional boundaries of the data space contains the center point c.* For $d = 2$ and $d = 3$ the lemma can be proved, however, it is definitely false for $d = 16$, as a counter example for it can be presented. Consider a point $p = (0.3, ...0.3)$ and a sphere of radius 0.7 centered at p. This sphere touches or intersects all 15 dimensional surfaces of the space but does not contain c. The Euclidean distance of p and c is $\sqrt{16 \cdot 0.2^2} = 0.8$ which is larger than the sphere's radius. This example illustrates that some simple properties which seem intuitive and are true for low dimensional data, do not hold as the data dimensionality grows.

The most basic effect of dimensionality is the exponential growth in volume. Consider a query consisting of a hypercube with selectivity 1% in the domain space of $[0,1]^d$. In order to keep the volume constant, the hyper-edge should increase exponentially. For example in the 2 dimensional space, the query width along each dimension, i.e. the hyper edge is $\sqrt[2]{0.01} = 0.1$, which is one tenth of the entire dimension. However with dimensionality as high as 40, the query width covers almost the whole range: $\sqrt[40]{0.01} = 0.84$.

As a result of the curse of dimensionality *Space partitioning* methods containing all *tree-based* approaches such as R-tree [Gut84] and K-D trees [Ben90], which perform very well when data dimensionality is not high, degrade to linear search for high enough dimensions [BGRS99]. The Pyramid [BBK98] and iDistance[YOTJ01] techniques map the high dimensional data to one dimension and partition/cluster that space to answer queries by translating them to the one dimensional space. An alternative to the space partitioning methods are the family of *hash-based* approaches which trade accuracy for efficiency, by returning approximate closest neighbors of a query point. In the following we describe the well-known *Locality Sensitive Hashing* (LSH) approach which we will use later to design efficient algorithms for the distributed version of this problem.

Locality Sensitive Hashing

The basic idea behind the *LSH-based* approaches is the application of *locality sensitive hashing* functions [GIM99]. A family of hash functions $H = \{h : S \rightarrow U\}$ is called (r_1, r_2, p_1, p_2)-sensitive if the following conditions are satisfied for any two points $\mathbf{q}, \mathbf{v} \in S$:

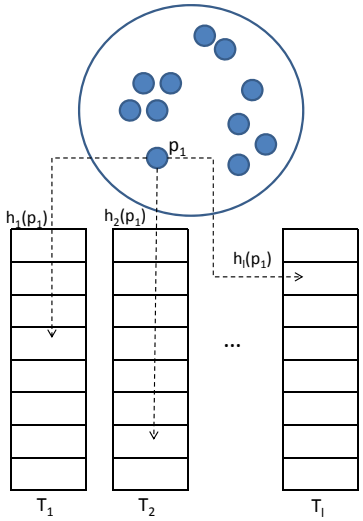


Figure 2.4: To improve the quality of search, multiple hash-tables are used in the LSH approach.

- if $dist(\mathbf{q}, \mathbf{v}) \leq r_1$ then $Pr_H(h(\mathbf{q}) = h(\mathbf{v})) \geq p_1$
- if $dist(\mathbf{q}, \mathbf{v}) > r_2$ then $Pr_H(h(\mathbf{q}) = h(\mathbf{v})) \leq p_2$

where S specifies the domain of points and $dist$ is the distance measure defined in this domain.

If $r_1 < r_2$ and $p_1 > p_2$, the salient property of these functions results in more similar objects being mapped to the same hash value than distant ones. The actual indexing is done using LSH functions and by building several hash-tables to increase the probability of collision (i.e., being mapped to the same hash value) for close points. At query time, the kNN search is performed by hashing the query point to one bucket per hash-table, scanning that bucket and then ranking all discovered objects by their distance to the query point. The closest k points are returned as the final result. Figure 2.4 illustrates the LSH approach, where l hash-tables are used.

While this method is very efficient in terms of time, tuning such hash functions depends on the distance of the query point to its closest neighbor. Several follow-ups of this method exist which try to solve the problems associated with it [BCG05, DIIM04, LJW+07, Pan06, AI06]. Very recently [APPK08] proposed a method based on distance hashing which has the advantage of not depending on any specific distance measure, but involves some off-line tuning of parameters.

Chapter 3

State of the Art

Ranking queries have been considered in a variety of environments in addition to traditional centralized settings. In top-k query evaluation over data streams, usually, a limited number of queries in form of monotonic aggregation functions are subscribed to the system and are continuously updated with relevant results. In this Chapter, we start first by describing different approaches for top-k query evaluation over data streams. Then we move to larger number of queries and higher dimensional data, by describing state of the art in *information filtering*. Finally, we discuss kNN queries over P2P networks, with an emphasis on the difficulty of kNN search over high dimensional data.

3.1 Top-k Evaluation and Join Processing Over Data Streams

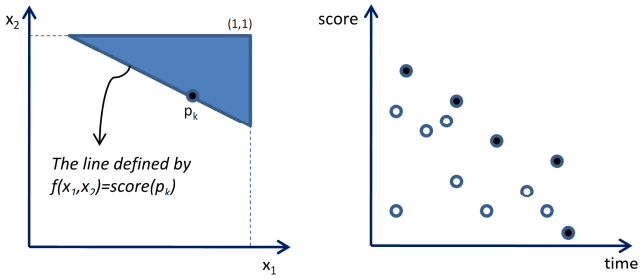
Top-k query evaluation over data streams in a sliding window model has been a hot topic in the previous years. Mouratidis et al. [MBP06] maintain a skyline [BKS01] which represents the possible top-k candidates, i.e., those items that have, due to the sliding window (timeouts), still a chance to get in the top-k result at some point. Their main focus is on the efficiency of the evaluations and do not consider space limitations. They assume all attributes of a data point are seen together which means exact score calculation is possible. We will describe this approach in more detail in Section 3.1.1. In a more general setting, [DGKS07] proposes indexing methods for answering ad hoc top-k queries utilizing arrangements. Complete information over object attributes is also assumed here. Also related to this research are continuous k nearest neighbors (kNN) queries on data streams, considered in [KOT04, BOPY07]. Koudas et al. present Disc [KOT04] for indexing high dimensional points using space filling curves to give approximate answers to kNN queries. On the other hand [BOPY07] considers a fixed number of queries and indices queries instead of incoming tuples in a structure similar to VA files [WSB98] to continuously provide exact answers in a sliding window model. A skyline of the tuples in the score-time space

is maintained to decide which tuples should be kept, therefore minimizing the needed storage. They also assume complete knowledge of the score once a tuple arrives in the stream. In [PZA08] the authors consider the top-k problem in a publish/subscribe context. The goal is to return the top-k most relevant publication to each subscriber. Similar to other approaches, they assume that each publication contains the complete information to calculate its score and rank against each subscription. In contrast, we consider a more general setting where the different attributes of an object appear in multiple non-synchronized streams, therefore the evaluation engine has partial information over the score of objects.

Another related problem considered in data stream processing is load shedding and approximate join processing. Das et al. [DGR03] consider approximate join processing in a sliding window model with limited resources. They propose an optimal off-line algorithm for evicting tuples when fixed amount of memory is available and two online algorithms: *PROB* which leverages the probability distribution of objects appearing in streams and *LIFE* which considers the lifetime of objects as well. [SW04] considers the same problem but also introduces the *age-based* model in which objects do not repeat in streams, therefore *PROB* is not applicable. [XYC05] generalizes the setting to stochastic streams. In [BSW04] the authors introduce the notion of k-constraints and exploit that to reduce the run-time state of continuous queries. Li et al. [LCKB06] exploit reference locality to reduce the cost of stream operators, such as joins. Processing multi-joins in a sliding window is considered in [GÖ03] where adaptations of nested loop and hash joins are proposed and evaluated.

3.1.1 Continuous top-k monitoring over sliding window data streams

Continuous top-k monitoring was first considered in [MBP06] and we will describe this approach in more details in the following. It is assumed that queries in form of monotonic aggregation functions are registered at the system and the goal is to keep them updated with their valid top-k results. Objects are considered valid while they belong to a sliding window. Arriving tuples contain all attributes of interest, such that as soon as an object arrives its score with regard to an aggregation function could be calculated. Assuming the data is d dimensional, a hyper-grid of d dimensions is used for storing the valid objects. The extent of each cell on every dimension is δ. For example in a two dimensional case, a cell c_{ij} contains all tuples with the first attribute in the range $[i \cdot \delta, (i+1) \cdot \delta)$ and second attribute in the range $[j \cdot \delta, (j+1) \cdot \delta)$. Also, as an efficient mechanism for evicting expired tuples, a time-sorted list is maintained where new tuples are inserted in the head and old data drop out of the tail. Since the queries are monotonic, an *influence region* can be defined for each. The influence region of each query, defined by its aggregation function and ranked k result, contains all cells in the grid, where if a tuple falls in that section it is a top-k result of this query. As an example consider the aggregation



(a) (b)

Figure 3.1: (a) The influence region of a query (b) The skyline points

function $f(x_1, x_2) = 2x_1 + 3x_2$ for query q. Also let the attributes of the data points to be normalized to $[0, 1]$ and let p_k be the object with the k highest score for q. The line $score(p_k) = 2x_1 + 3x_2$ divides the space into two parts. As shown in Figure 3.1(a), the shaded area represents the influence region of q, i.e., any update falling in the influence region affects the top-k results of q.

For each cell c in the grid, a list of queries whose influence region contains c are also maintained. When a new tuple arrives, its position in the grid is determined. Assume it falls in cell c. Then only the queries stored in cell c need to be updated. Maintaining this information allows the system to avoid processing all queries in the system when a new tuple arrives. On the other hand, when an object which is in the top-k result set of a query expires, that query should be re-evaluated. To avoid re-evaluation each time an object expires, the concept of *skylines* [BKS01] is used. The skyline of a set with regard to a given aggregation function contains those objects which have a chance of becoming a top result, as other objects expire. Each point in the dataset is considered in the score-time space, where score corresponds to its aggregated value with regard to the query aggregation function and time denotes its expiration time. We say a point dominates another one, if it has both a better score and a longer life time. The skyline of a set consists of points which are not dominated by any other point. Figure 3.1(b) illustrates the points in the skyline as shaded circles.

With the above mentioned computational and maintainable modules, continuous monitoring of queries, where data points arrive completely in the stream and are of low dimensionality, can be done very efficiently. However, as we will see in Chapter 4, the above solution is not applicable to the case where multiple non-synchronized streams report different attributes of an object.

3.2 Information Filtering

In information filtering, a stream of incoming text documents, from different sources, is considered and users subscribe to a system with their favorite profiles and receive notification whenever a relevant document arrives at the system. Traditionally, relevance is defined by a fixed threshold or a boolean model

is used, therefore the queries could essentially be considered as iceberg queries over a data stream. Two main features differentiate information filtering from top-k processing over streams as described in the previous Section. First, the number of queries considered in an information filtering system is usually much larger, and second, since the incoming data is in form of text, it has high dimensionality as compared to usually low dimensional data considered in other scenarios. When the number of queries which should be constantly monitored is large, *query indexing* is a more appropriate approach compared to the usual *data indexing*. Information filtering or *selective dissemination of information* is considered dual to classical information retrieval. Information filtering has been considered in vector space [BM96, YGM94a], boolean [YGM94b], and more recently *AWP* [TKD09] models. In [YGM94a], a profile filtering scheme is proposed which is based on distinguishing the *significant* and *insignificant* terms of a profile based on the given threshold. In [BM96] the previous method, which reduces the cost of disk I/O at the expense of larger indices, is combined with a document batching process which takes advantage of the sparsity of the profile and document matrices and writes the partial similarity matrix to disk, improving the efficiency. Callan [Cal96] describes a statical document filtering system based on the inference network model of information retrieval.

Information filtering is essentially similar to bichromatic reverse nearest neighbor search (RNN). Given a database of points P, a set of query points Q, and a similarity measure between the members of P and Q, in bichromatic RNN search, with a query $q \in Q$ the goal is to find $p \in P$ which is closer to q than any other point of Q. Most proposed methods for this problem consider two dimensions. In [KM00] two separate R-trees are used as the index structure for RNN search. [SAA00] considers the monochromatic version of RNN in two dimensions and is based on the geometric observation that the maximum number of RNN's in two dimensions for a query point is 6. Singh et al. [SFT03] propose an approximate method for RNN search in high dimensional data which first finds the NN's of a query point with the hope that its RNN is actually among them.

In the following we describe the approach in [YGM94a] and [MP09] which are the closest to our problem considered in Chapter 5.

3.2.1 Threshold-based Information Filtering

In [YGM94a], motivated by the wealth of information generators and in order to assist users in being informed of interesting content, an information filtering mechanism is introduced. Large number of users subscribe their interests as *profiles* to the system and get continuously updated with new relevant documents. The documents and profiles are assumed to follow the vector space model, i.e., each document or profile is described by a set of weighted terms. The degree of similarity between a document and a profile is measured by the well known *cosine* measure.

Given a threshold value T, when a new document d arrives, all profiles

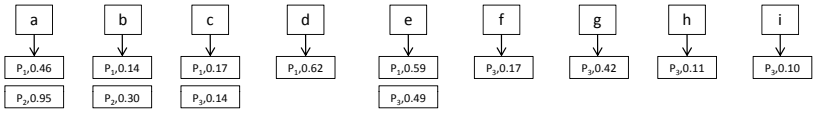


Figure 3.2: An example of the PI indexing method

which have a larger similarity degree than T to this document are informed of d. The brute force approach, compares each existing profile to a new document and updates the affected profiles. However, such a scheme is not scalable to large number of users and high input rates. Therefore a Profile Indexing (PI) method is described in [YGM94a]. Similar to traditional IR, inverted lists are constructed for each term, but this time the lists contain the profiles registered at the system. For each term t, all profiles which have a non-zero weight for t are collected. The inverted list for a term t is made of *postings*, where each posting consists of a profile identifier and the weight of t in that profile. Figure 3.2 shows an example where there are nine terms and three profiles. When a new document arrives, only those lists corresponding to a non-zero weighted term in the document are processed against this document. Lists are processed one by one and a score value is kept for each profile seen in a list and updated when observed in subsequently processed lists. Those profiles whose score is larger than T are then updated with d.

In the PI method, a profile is indexed by all its terms. As an alternative, the authors introduce the Selective Profile Indexing (SPI) approach, which indexes each profile by a selected number of its terms. This selectivity is based on the definitions of *significant* and *insignificant* terms. An insignificant subset of terms for a profile p , is a subset of p's terms which if are the intersection of p and a document, do not conclude in a similarity score larger than T between that document and p. The most insignificant terms of a profile with regard to a given threshold can be identified with a simple greedy algorithm. At index time, a profile is indexed by its significant terms only and the insignificant terms are repeated as an extension each time a profile appears in the inverted list of a term. This ensures that the exact score of a profile with regard to a document can still be calculated. Figure 3.3 illustrates the SPI approach where $T = 0.20$. The SPI method has larger space overhead compared to the PI approach, as the insignificant terms are repeated with every significant term being indexed. However the SPI method results in better processing time, as the number of profile comparisons per document is less than PI.

In Chapter 5, we study a similar problem. However, we argue that setting the threshold to a fixed value does not perform well in a highly dynamic setting. As we do not consider the threshold values to be fixed, the above solution is not applicable to our problem.

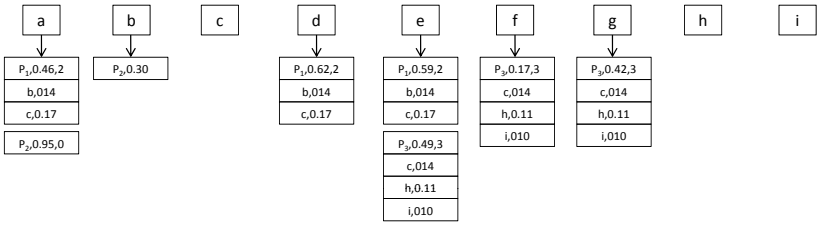


Figure 3.3: An example of the SPI indexing method where T=0.20

3.2.2 Information Filtering With Continuous Monitoring of Large Cardinality Top-k Queries

The closest work in the literature to the problem we consider in Chapter 5 is the recent work by Mouratidis et al. [MP09] which considers processing continuous text search queries. Queries in this setting are assumed to consist of weighted terms. The incoming documents are also processed to be represented similarly and a monotonic aggregation function is used to measure the similarity of each document to a query. Valid documents are maintained in an inverted list which allows for a threshold algorithm to be executed as needed. The main goal is to avoid the brute-force method of comparing each incoming document with all queries registered in the system. While this is similar to our setup in Chapter 5, the key idea in this approach is to keep the state of the TA algorithm, for each query, in a per-term index organized as a tree. Upon arrival of new documents, the tree is scanned for all potentially affected profiles and in case of a change in the score of the document at rank k, the thresholds are updated upwards. In the case of the removal of old documents the index lists' scan lines will be adapted downwards. The rationale behind this continuous adaptation of the scan lines is that tight bounds cause fewer documents having to be evaluated against the registered queries. However, these scan lines are too often not a good (tight) description of the actually more interesting score at rank k, leading to the problem that many profiles have to be checked for modification with almost every incoming document. An additional effect of the the scan line based indexing is the large number of potential candidates held in the resultset. We address both problems (profile indexing and result maintenance) in this work and have implemented the approach by Mouratidis et al. [MP09] and include it in our experimental evaluation.

3.3 Distributed Nearest Neighbor Search

In kNN search (c.f., Section 2.2.2), given a query in form of the points in a set and a similarity measure, the goal is to return the k closest points to the query point. With the emergence of the P2P paradigm [SMK$^+$01, RFH$^+$01] and distributed large data sets, there has been a tendency to leverage the power of distributed computing by sharing the cost incurred by kNN search over a set

of machines. A number of P2P approaches, such as [CLM+07, BAS04, BB05] have been proposed for similarity search, but they are either dedicated to one dimensional data or do not consider very high dimensional data. MCAN [FGZ05] uses a pivot-based technique to map the high dimensional metric data to an N-dimensional vector space, and then uses CAN [RFH+01] as its underlying structured P2P system. The pivots are chosen based on the data, which is preprocessed in a centralized fashion and then distributed over the peers. SWAM [BKS04] is a family of *Small World Access Methods*, which aims at building a network topology that groups together peers with similar content. In this structure peers can hold a single data item each, which is not well-suited for large data sets. SkipIndex [ZKW04] and VBI-tree [JOV+06] both rely on *tree-based* approaches which do not scale well when data dimensions are high. In pSearch [TXD03], the well known information retrieval techniques *Vector Space Model*(VSM) and *Latent Semantic Indexing* (LSI) are used to generate a semantic space. This Cartesian space is then directly mapped to a multi-dimensional CAN which basically has the same dimensionality of the Cartesian space (as high as 300 dimensions). Since the dimensionality of the underlying peer-to-peer network depends on the dimensionality of the data (or the number of reduced dimensions), different overlays are needed for various data sets with different dimensionality. This dependency and centralized computation of LSI make approach less practical in real applications. In [SEAA04] the authors follow pSearch by employing VSM and LSI, but map the resulting high dimensional Cartesian space to a one dimensional Chord ring. Unlike pSearch, this method is independent of the corpus size and dimensionality. This is the closest work in state of the art to our considered problem, since it considers high dimensional data over a structured P2P system, for a more detailed description of this work see Section 3.3.1. Recently, SimPeer [DVKV07] was proposed, which uses the principle of iDistance [JOT+05] to provide range search capabilities in a hierarchical unstructured P2P network for high dimensional data. In this work also, the peers are assumed to hold and maintain their own data. On the contrary, we consider efficient similarity search over structured P2P networks, which guarantees logarithmic lookup time in terms of network size, and leverage on *LSH-based* approaches to provide approximate results to KNN search efficiently, even in very high dimensional data. Our approach also enables efficient range search which is difficult in LSH-based approaches.

3.3.1 The Approach by Sahin et al.

In [SEAA04] the authors consider kNN search over high dimensional data in a structured P2P network. Given a set of high dimensional points, the goal is to distribute the data in the P2P network, such that efficient kNN search is possible and a fair load balance ensures the robustness of the network. A globally known list $R = r_1, r_2, ..., r_v$ of reference data points is considered. These are either randomly chosen from the data set or are the cluster representatives of a clustered sample set. In order to index a data point v, it is compared against

all reference points and a sorted list of references in increasing distance to v is constructed. The first j references, which are the reference points closest to v are used to make the Chord key for it: the binary representations of the ID's of these j references are concatenated, with the highest relevant reference as the high order bits. If there are any remaining bits in the Chord bit representation, they are filled with zeros. The intuition behind this approach is that points which are close to each other, share common top references and will therefore be stored at the same peer. In order to increase the probability of this event, multiple Chord keys are made for each data point, choosing j different reference points, or different permutations of them. At query time, the query point is similarly mapped to the Chord ring and the corresponding peers are scanned with the k closest points returned and merged at the peer issuing the query. Figure 3.4 shows an example of indexing a data point. The number of references is 7, the Chord range is $(0, 2^{10})$, j is 2 and each data point is replicated three times.

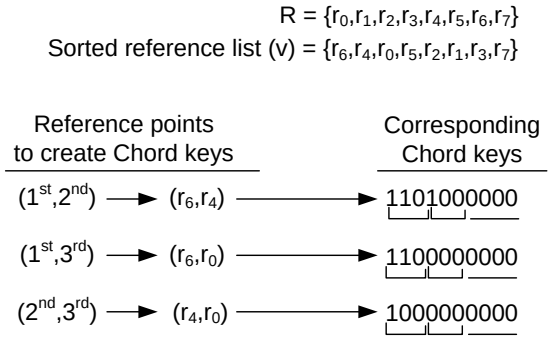


Figure 3.4: Illustration of mapping a point to the Chord peer space by Sahin et al.

This approach of mapping data points to the Chord ring focuses on placing close points on the same peer and does not exploit nearby peers. For example a data point q close to v might have (r_6, r_4) as its two first closest references, corresponding to the Chord key 1001100000, resulting in placing it on a far peer from where v is placed (*i.e.* 1101000000). In our approach we aim at placing close points on the same peer *or* neighboring peers to exploit the linear order of peers in Chord style DHTs and avoid high number of DHT lookups.

Chapter 4

Evaluating Top-k Queries over Incomplete Data Streams

4.1 Introduction

Research on data streams has gained a lot of interest in the past few years [BBD+02, Mut05, CKT08, MBP06, KOT04, BOPY07]. Many data and query characteristics of modern applications are best captured by this computational model, where data streams in continuously at high rates and each tuple has the chance of being observed only once. Memory constraints usually force the eviction of old tuples to let new tuples arrive and be processed. In this Chapter, we consider tracking top-k items over multiple data streams in a sliding window. Each stream represents one particular dimension of interest, for instance, a particular attribute of the observed item w.r.t. a sensor location. We address a broad area of application scenarios, like network monitoring and sensor network data processing such as observing local natural phenomena, a common task in environmental sciences.

While top-k processing over sliding window data streams has been the focus of several recent papers, [BOPY07, MBP06, JY+08, DGKS07] (c.f., Section 3.1), all these works assume complete information over the arriving object's attributes or desired scores. In this model tuples arrive in *one* stream where all attributes of the object have been measured and projected in this tuple, or, different attributes arrive in *several* streams but with an *unlikely* assumption that all attributes of an object arrive at the same time in all streams. Therefore, in this model it is possible to calculate the exact score of each object with regard to a desired query instantly as the object arrives. On the contrary, we observe that in many streaming scenarios this is not the case.

For example, consider an Internet Service Provider that monitors traffic at different routers in a network. Each router sends detailed traffic logs of different

flows to a central server. These logs can consist of the source and destination IP addresses, the number and size of packets corresponding to them, and a time-stamp. The central server receives this information from different routers (i.e., in multiple streams) and can process various queries to better estimate and control traffic over the network or prevent security attacks. Top-k queries are commonly used in these applications, e.g., to *continuously report the top-20 flows with largest total size*. Since each flow can appear in several routers at different time-stamps, the exact score of each flow (in this example sum of *size*s measured in all routers) can not be computed unless a tuple representing this flow arrives in *all* of the streams.

As another example, imagine a network of cameras on highways with detectors of the number plates, each reporting the observed vehicles in a stream to a central unit. A query would involve certain camera streams in a given area and the goal is to report based on (time-stamp, plate-number, speed)-triples the fastest drivers. Naturally, a vehicle is captured by different cameras at different times, so its attributes (i.e., speed at various points) arrive at the central unit with some delay. Not *all vehicles* are observed by *all cameras*, as there are various alternative paths. In similar scenarios where moving objects are tracked by stationed sensors, for example in supply chain management, the objects are incomplete by nature as they do not necessarily pass through all sensors, therefore do not explicitly possess values for all attributes. Unless all attributes of an object have been observed, or sufficient time has passed since its last observed attribute, it is not certain that the object's score will not change. Additionally, this incomplete view can be due to measurement noises, lossy transmissions, or delayed arrivals as a result of network characteristics, as opposed to the nature of attributes and monitored objects. Environmental monitoring scenarios can serve as an example to this.

The sliding window model adds up to this uncertainty as different attributes of the same object may be valid for different amounts of time. As a result, a system favoring exact results should maintain several aggregation scores, with regard to different expiration times of its attributes. This, significantly changes the properties of the system, especially with regard to the necessary storage.

Motivated by these scenarios, where complete observations are rare, we choose incomplete streams as our underlying streaming model. We show that extensions of existing approaches to the uncertain score scenario do not perform well in practice, particularly with regard to storage which poses fundamental restrictions in stream processing engines.

4.1.1 Problem Statement and System Model

Let $O = \{p, q, ...\}$ be the set of objects we are monitoring. We consider d incoming streams $s_1, s_2, ..., s_d$, each corresponding to one attribute of the objects. Note that we can similarly consider merging all streams to one incoming stream and supplement each tuple with the attribute it is describing. However, for the sake of simple presentation we will use the multi stream notation through out

this Chapter. Each stream s_i contains tuples of the form $\langle p.id, p.value(i), p.t_i \rangle$, where $p.id$ uniquely identifies p, $p.value(i)$ is the value of attribute i of p and $p.t_i$ is the arrival time of this tuple. We assume all attribute values $p.value(i)$ are normalized to $[0, 1]$. Objects do not necessarily appear in all streams and can arrive in different streams at different time-stamps. Tuples continuously stream in and they are considered *valid* while they belong to a sliding window W. Sliding windows can be either count or time based. Our algorithms can naturally handle both kinds of sliding windows. For simplicity, we assume each object appears in each stream at most once. We later show how our methods are extendable to the case where this assumption is not valid.

At each instant of time, we can calculate the scores of *valid* objects given a monotonically increasing aggregation function:
$score(p) = f(p.value(1), ..., p.value(d))$. An object is considered valid if it has at least one valid attribute. In calculating $score(p)$, the value of an unseen or expired attribute is considered to be the smallest possible; in our case where values are normalized to $[0, 1]$ this is 0. The score of an object can increase over time as some of its unseen attributes arrive or it can decrease as some of its attributes expire.

Given the above, we are interested in continuously monitoring the top-k valid objects with regard to their scores. We consider the tradeoff of minimizing storage consumption and result accuracy. Our goal is to sum-up two contradictory conditions: keep less than necessary, but maintain the accuracy of the top-k results.

Figure 4.1 shows an example of our model in the network monitoring scenario. In this case, objects are *flows*, therefore *id* can be the concatenation of the source and destination IP addresses. Attribute i of each flow is its size measured at router i. Objects do not arrive in the same order in different streams. The aggregation function is sum and a time based window of size 100 is assumed. As can be seen, the score of objects increase and decrease over time as new attributes arrive or old ones expire.

This Chapter is based on our work in [HMA09b] and is organized as follows. Section 4.2 presents two exact algorithms and discusses the new notion of dominance in order to enable retaining only the necessary objects for providing exact results. The increase in memory consumption is also discussed in this Section which motivates our approximate algorithm. Section 4.3 presents our approximate approach to deal with partial knowledge imposed by the incomplete data streams. Section 4.4 shows how our proposed algorithms can be extended to the case where each object may appear several times in each stream. Section 4.5 discusses this problem when a *fixed* amount of memory is available and presents an optimum off-line solution. Section 4.6 presents the experimental evaluation. Section 4.7 concludes this Chapter.

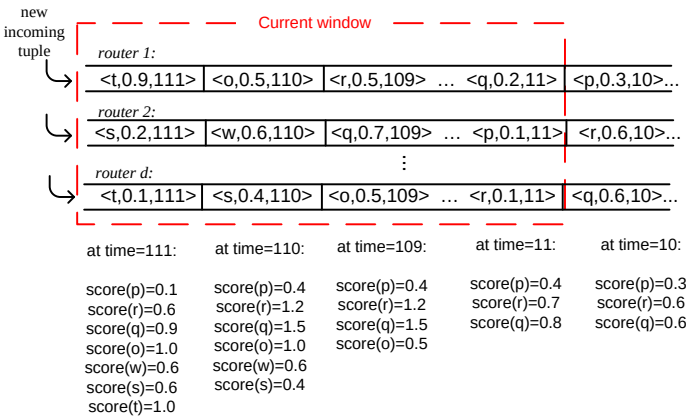


Figure 4.1: An example: streams are generated at various routers and flows are the monitored objects

4.2 Exact Algorithms

We describe two exact algorithms for evaluating top-k queries over incomplete data streams. The first algorithm is an adaptation of the threshold sorted list mechanism used in traditional databases. The idea is to store the tuples of each stream in a sorted list and use the Threshold Algorithm (TA) [FLN03], as described in Section 2.2.1, in order to evaluate top-k queries. The second algorithm builds on early aggregation of tuples as they stream in. It enables pruning of tuples which do not have a chance of becoming a top-k result and can be safely dropped.

4.2.1 Sorted List Algorithm (SLA)

We assume all valid tuples are sorted in a first-in-first-out list. This provides an efficient mechanism for evicting expired tuples. Newly arriving tuples in each stream are placed at the head of this list and old tuples are dropped from the tail. Note that this is applicable to both count-based and time-based sliding windows. In addition to this list we maintain d sorted lists, one per stream (i.e., for each attribute). Upon receiving $\langle p.id, p.value(i), p.t\rangle$ from the i^{th} stream, $(p.id, p.value(i))$ is inserted in the i^{th} list which is sorted based on the *value* field. When a tuple expires, it is also removed from the sorted list it belongs to.

In order to evaluate a top-k query, the TA algorithm is used. Similar to [MBP06] we use the technique of [YYY+03] for efficient maintenance of the top-k results in face of frequent insertions/deletions: we maintain $k_{max} > k$ entries for a top-k query in order to reduce re-computations. When a new tuple $\langle p.id, p.value(i), p.t\rangle$ arrives, p's new score is computed by random access to all other attribute lists. The result list is updated accordingly: if p's score is higher than the least score in this list, p is inserted to the result view. Similarly whenever a tuple expires, the score of its corresponding object decreases. If this

object was part of the result view, the result view is updated. Once the size of the result view falls below k the TA algorithm is called to recompute the top-k results.

4.2.2 Early Aggregation Algorithm (EAA)

In monitoring scenarios where tuples stream in with very high rate, it is often impossible to save all incoming data as in the previous approach. Most of the tuples are not interesting and can be dropped. Therefore with limited resources, only objects which have a chance of becoming a top-k result should be saved. These include those objects which at their time of arrival are not among the top-k results of the query, but over time as some objects expire qualify as top-k results. This idea is used in [BOPY07, MBP06, MP07] to devise efficient methods for processing fixed kNN and top-k queries. In the following we first describe how such objects are identified when instant score evaluation is possible and then show how this can be extended to our model.

If the data arrives in a way that allows for a full evaluation instantly, i.e., all attributes of an object arrive at the same time, the score of each object can be calculated with regard to a fixed query (i.e., a given aggregation function), and this score does not change during the object's life time. Now each object p can be regarded as a point in the score/time space represented by two attributes, its score: $p.score$, and time of arrival: $p.time$. A point p is said to *dominate* another point q if and only if p is preferable to q in all attributes. For our problem this translates to $p.score > q.score$ and $p.time > q.time$. A point is in the *k-skyband* of the dataset if it is dominated by less than k other points in the dataset. *Skyline* constitutes the case for $k = 1$. It is easy to observe, and has been formally proven in [BOPY07, MBP06, MP07], that it is sufficient to save only points in the k-skyband over these two attributes of the dataset to answer top-k or kNN queries. This is also the minimum number of points required to answer the top-k query *accurately*.

It should be noted that although the objects are originally d dimensional, the skyband is calculated over only two attributes (score, which is an aggregation of the initial d attributes, and time of arrival).

The shortcoming of these approaches is that they assume the data to arrive in a way that allows for a full evaluation instantly. In case of multiple non-synchronized data streams, the score of an object may change over time as more of its attributes are observed or some expire. As a result, the classic dominance check can not be used to drop objects. However, since we are assuming a monotone aggregation function we can calculate the highest score an object can acquire. This is a standard concept in top-k query processing [FLN03] that allows for pruning items based on their upper bound score. This upper bound can be used in performing a conservative dominance check, which considers possible increases in the score of an object. Nevertheless, as some attributes of an object expire before the rest, its score can also decrease over time, which means the dominance relation between two objects may change over time. We

solve this problem by creating several instances of the same object with different expiring times in such a way that the score of these instances can only increase over time. In the following we first explain how these aggregated instances are created and then show how the dominance check condition should be changed in this case such that all objects having a chance of being among the top-k are still in the k-skyband.

Instance Creation

Assume the attributes of an object p have been observed in a subset $I = \{i_1, i_2, ..i_r\} \subseteq \{1, .., d\}$ of streams and w.l.o.g. $p.t_{i_1} < p.t_{i_2} < ... < p.t_{i_r}$. We create r aggregated instances of p in the following way:

$$p^j = \langle p.id, p^j.currentscore, p^j.bestscore, p^j.t \rangle$$

where $p^j.currentscore = f(v_w(1), ..., v_w(d))$ and

$$v_w(x) = \begin{cases} p.value(x) & x \in I \wedge p.t_x \geq p.t_{i_j} \\ 0 & otherwise \end{cases}$$

$p^j.bestscore$ represents the highest score this object can get during its life time and is calculated as $p^j.bestscore = f(v_b(1), ..., v_b(d))$ and

$$v_b(x) = \begin{cases} p.value(x) & x \in I \wedge p.t_x \geq p.t_{i_j} \\ 0 & x \in I \wedge p.t_x < p.t_{i_j} \\ 1 & otherwise \end{cases}$$

where the second condition is based on the fact that each object is observed once per stream. The arrival time of this instance is set to $p^j.t = p.t_{i_j}$, which is the earliest time among all observed attributes which are considered for calculating its *currentscore*. This ensure that *currentscore* can only increase during this instance's life time. If a new attribute i_{r+1} of p is observed, the above instances are updated accordingly: $v_w(i_{r+1}) = v_b(i_{r+1}) = p.value(i_{r+1})$, which results in a larger value for *currentscore* and a smaller one for *bestscore*. As a result, the interval of [*currentscore, bestscore*] can only shrink over time for each aggregated instance. Also a new instance p^{r+1} is created accordingly. Figure 4.2 shows an example of object instances and their score intervals in case of $W = 100$. p's second attribute is observed earliest at $t = 1$, where p^1 is created. When p's third attribute arrives in stream 3, p^1's scores are updated accordingly and p^2 is created. As can be seen the score interval decreases: $p^1.currentscore$ increases while $p^1.bestscore$ decreases. Upon observing the first attribute, the evaluation engine has full information over the score of this object. p^1 and p^2's scores are updated and p^3 is created.

The introduction of currentscore and bestscore follows the scoring introduced by Fagin et. al. [FLN03] in their NRA algorithm that uses only sequential scans of the input data. However, in NRA the best possible score to be considered when calculating the upper bound score (bestscore) decreases with ongoing sequential scan, as the input data per attribute is considered to decrease in score,

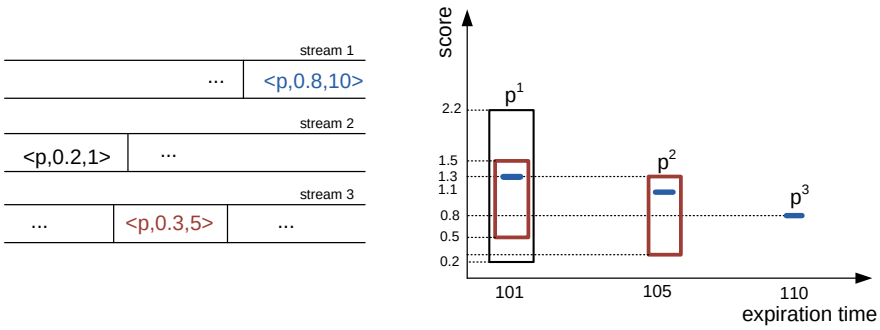


Figure 4.2: An example of object instance creation

which would be an unrealistic assumption in the streaming data scenario we consider.

Interval Dominance

Given a set S of such object instances we are interested in keeping only those which have a chance of becoming a top-k in future. We define the interval dominance as follows.

Definition 1 *[Interval Dominance] Given two object instances p^i and q^j we say p^i dominates q^j and denote this by $p^i \succcurlyeq q^j$ iff $p^i.t > q^j.t$ and $p^i.currentscore > q^j.bestscore$.*

The **k dominance set** of S, denoted as S_k, consists of all instances which are dominated by less than k other distinct object instances. Two object instances are said to be *distinct* if they possess non equal *id*s. Interval Dominance has the following desirable properties:

P1-Persistence: Dominance is persistent during an instance's life time: if $p^i \succcurlyeq q^j$ at time τ, $p^i \succcurlyeq q^j$ for all times $t > \tau$. This is because $p^i.currentscore_\tau > q^j.bestscore_\tau$ and *currentscore* can only increase over time while *bestscore* can only decrease, so $p^i.currentscore_t > q^j.bestscore_t$ for $t > \tau$, where $p^i.bestscore_t$ denotes p^i's bestscore at time t. Also the *time* attribute t of instance objects does not change over time.

P2-Transitivity: If $p^i \succcurlyeq q^j$ and $q^j \succcurlyeq r^k$ then $p^i \succcurlyeq r^k$. The definition of interval dominance directly results in this property. As a result, if $p^i \succcurlyeq q^j$ and $p^i \notin S_k$ then also $q^j \notin S_k$.

Theorem 1 *Assume S is the set of all valid instances. It is necessary and sufficient to retain the object instances in S_k to provide exact top-k results.*

For the proof see the appendix.

Structure and Maintenance

Our *Early Aggregation Algorithm* (EAA) is based on maintaining the k dominance set of valid instances as new tuples arrive and old ones expire. We keep the list of valid objects in a hash-table based on their *id*s. As described previously in the Instance Creation Section, for each valid object we have r aggregated instances where r is the number of valid attributes with distinct arrival times. We keep a sorted list of these instances based on their *time* attribute and keep pointers to them from the corresponding hash-table entry. Upon arrival of a new tuple $\langle p.id, p.value(i), p.t\rangle$ from the i^{th} stream, we first look up $p.id$ in the validity hash-table. If $p.id$ exists in the hash-table, a number of other attributes of p have been observed before. Assume r distinct attributes were observed for p before the new tuple. We have created r instances of p as described previously. These instances are updated by taking $p.value(i)$ into account instead of 0 in calculating *currentscore* or 1 in calculating *bestscore*. We also create a new instance p^{r+1} as described in Section 4.2.2 and insert it in the *time* sorted list. A pointer to this new object is also kept in the validity hash-table in the corresponding entry.

So far we described the insertion of new tuples and creation/modification of object instances. In order to prune unnecessary instances, we maintain the number of objects which dominate each instance. We keep a dominance counter *dc* for each instance, which shows the number of distinct instances that dominate this instance, and maintain this value during the instance's updates. To facilitate maintaining *dc* and avoid scanning all instances each time an update occurs, we keep also two sorted lists of score values: one for *bestscores* denoted by *bsorted* and one for *currentscores* shown by *csorted*. For an object instance p^i, we create two entries of the form $\langle id, v\rangle$ with the following values: $\langle p^i.id, p^i.currentscore\rangle$ and $\langle p^i.id, p^i.bestscore\rangle$ and insert them in the *csorted* and *bsorted* lists respectively. Figure 4.3 illustrates the data structures we use.

The dominance counter of each instance can be calculated utilizing the three described lists. When p^1 is created, it has the last expiration time, as its time attribute t is set to the arrival time of the last received tuple. So it can not be dominated by any other instances. However, it dominates those instances whose *bestscore* is less than $p^1.currentscore$. These instances are easily accessed by first identifying the position of $p^i.currentscore$ in *bsorted*, which is possible by a binary search in that list, and from that point scanning all entries in a descending order and increasing their *dc* values. The steps necessary for this operation are shown in Algorithm 1. For each instance object we also keep a list of *id*s of instances which dominate this one: *dcids* and use it to avoid recounting non-distinct dominating instances.

Whenever an object instance p^i is modified (due to the observation of a new attribute of its corresponding object) it may dominate more instances, as its *currentscore* increases, at the same time it may be dominated by more instances, since its *bestscore* has decreased. The two score sorted lists are utilized to identify and update these involved instances. We only need to check an instance q^j, if $q^j.bestscore$ is larger than p^i's old *currentscore* and is smaller than its new

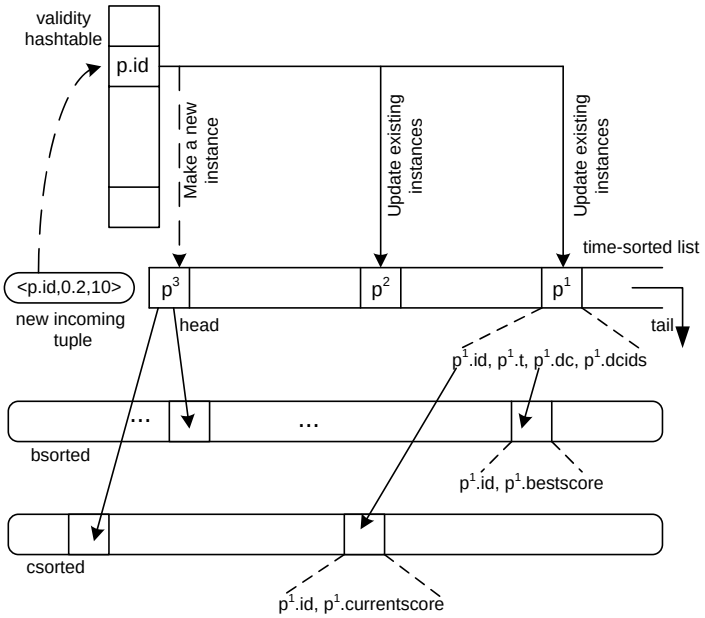


Figure 4.3: Structures for maintaining k-skyband

value. Such instances are easily identified by looking up p^i's old *currentscore* in *bsorted* and performing a scan from the found position in an ascending order until the value of the entry is larger than $p^i.currentscore$. We perform similarly for p^i's *bestscore*, looking its old value up in *csorted* and scanning in descending order until the value of the entry is smaller than $p^i.bestscore$. Algorithm 2 shows the details.

The object instances whose dominance counter *dc* hits k can be safely discarded. They are removed from the three sorted lists and their corresponding pointer is also eliminated from the validity hash-table. Note that removal of such instances does not effect the dominance counters of other instances. This is due to the *transitivity* property of dominance: all those instance which were dominated by p^i are also dominated by instances which dominate p^i. As a result if p^i is removed because it is dominated by k distinct instances, all other instances which were dominated by p^i have been removed before (because they were dominated by k instances earlier). Also objects which expire fall off the tail of the time sorted list and are also removed from the two score sorted lists and the validity hash-table. Their removal also does not effect other instances, as all instances which they could have dominated have expired before.

The top-k elements are identified by scanning the *csorted* list in a descending fashion until *currentscore*s of k distinct instances are identified. Similar to the Sorted List Algorithm (SLA) we can materialize k_{max} instances to avoid a high frequency of evaluations, though due to the availability of the score sorted list they are much faster than the evaluations done in the SLA algorithm.

```
Input: p.id,p.value(i),p.t,tsorted,csorted,bsorted
r=1;
if validityhashtable.contains(p.id) then
    forall E=validityhashtable.entry do
        update(E,p.value(i),tsorted,csorted,bsorted) ;
        r++;
    end
end
p^r=createInstance(p.id,p.value(i),p.t,r); tsorted.append(p^r,0,null);
csorted.insert(p.id,p^r.currentscore); bsorted.insert(p.id,p^r.bestscore);
foreach object instance q in bsorted do
    if q.bestscore < p^r.currentscore and p^r.id ∉ q.dcids then
        q.dc++;
        q.dcids.add(p.id);
    end
end
```

Algorithm 1: EAA steps for inserting a new tuple

On Skies and Linear Growth

It has been shown in [BKST78] that for a set of d dimensional objects of size n, under uniform and independent attribute selection, the skyline size is $O((ln\ n)^{d-1})$. As a result, in case of synchronous streams when exact score computation is possible, the number of objects which should be stored to provide exact top-k evaluation, is $O((ln\ n))$, as the skyline is computed over two attributes: score and time. In the following we show that with the interval dominance check, the size of the dominance set grows linearly in the size of the original set. As a result, although the expected storage consumption of EAA is less than SLA, still, it is linear in size of W. As we will see later, this gives the basis for our proposed approximate algorithm which aims at minimizing the memory consumption.

Lemma 1 *[Linear Growth] Given a set S consisting of n elements of the form $\langle currentscore_i, bestscore_i, time_i\rangle$, where $currentscore_i$ are chosen uniformly at random from $[0,1]$ and $bestscore_i = min(currentscore_i + \epsilon, 1)$. The* **expected size of the dominance set** *S^* is in $\Omega(n)$ where the constant depends on ϵ.*

For the proof see the appendix. Figures 4.4 and 4.5 report on the results an experiment to analyze the behavior of the skyline size with changing dataset size and changing the distance between best and current scores (i.e., the interval size). We run the experiments for uniform and skewed data. For skewed data we use an exponential distribution with parameter 1.5 and place the skew once on smaller values (minimum) and once on larger values (maximum). In exponentially distributed data when the skew is more towards smaller values we observe that the skyline size is similar to the standard skyline size, where only the current scores are taken to account (logarithmic in the dataset size).

```
Input: E,p.value(i),tsorted,csorted,bsorted
p^r=E.getInstance;
oldcurrentscore= p^r.currentscore;
oldbestscore = p^r.bestscore;
p^r.updateCurrentscore(p.value(i));
p^r.updateBestscore(p.value(i));
foreach object instance q^j in bsorted do
    if oldcurrentscore < q^j.bestscore < p^r.currentscore then
        if p^r.t > q.t and p^r.id ∉ q^j.dcids then
            q^j.dc++;
            q^j.dcids.add(p^r.id);
        end
    end
end
foreach object instance q^j in csorted do
    if p^r.bestscore < q^j.currentscore < oldbestscore then
        if q^j.t > p^r.t and q^j.id ∉ p^r.dcids then
            p^r.dc++;
            p^r.dcids.add(q^j.id);
        end
    end
end
```

Algorithm 2: EAA steps for updating an existing instance object

However, data is skewed towards larger values, we observe a linear trend similar to uniformly distributed data. Figure 4.5 shows the effect of interval size on the skyline size when keeping the dataset size constant. We observe linear growth while increasing the interval size for uniform data as well as for exponentially distributed data with a skew towards larger values. The experiment were conducted 50 times and we report on the average size values.

4.3 Score Estimation Based on Appearance Correlation

As seen in the previous Section, the number of objects which should be retained in order to provide exact answers to a top-k query in our streaming model is linear in the size of the sliding window. Such storage may not be available, especially when objects are stored in main memory to enable fast and online evaluation of the top-k query. Returning approximate query answers instead of exact answers is a graceful way of dealing with limited resources and has been applied in many streaming problems before [DGR03, XYC05, SW04]. In this Section we describe an algorithm which uses smaller storage at the expense of providing approximate results as opposed to exact ones. Our algorithm can be

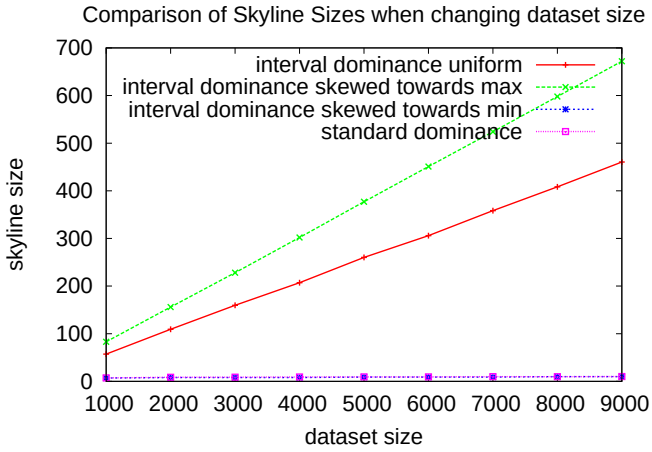


Figure 4.4: Skyline size when changing the data set size

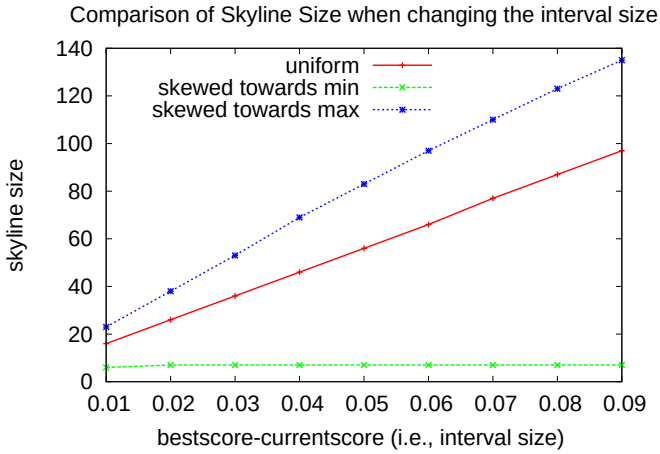


Figure 4.5: Skyline size when changing the interval size. Dataset size is fixed to 1000.

seen as a *semantic load shedding* algorithm which aims at maximizing the quality of the reported top-k. As in other classical work concerning top-k evaluation our measure of quality is the average precision obtained during all evaluations.

As shown in the proof of Lemma 1 and observed in Figure 4.5, the size of the dominance set is directly influenced by the score interval size: ($bestscore - currentscore$). Our approximate scheme leverages this fact to provide better space efficiency. *bestscore* for each object instance gives an upper bound of the score this instance can acquire during its life time. Previously, we considered the maximum value for unseen attributes in calculating *bestscore*. However, in most streaming scenarios that are best captured under our non-synchronized multiple stream model such as network or vehicle monitoring, not all attributes of an object are observed. This conceptually means that the value considered for such attributes in evaluating the object's *bestscore* can be set as the minimum value an attribute can get. Pairs of streams usually have different correlations among them. For example in a vehicle monitoring scenario, cars which are observed in path i maybe more likely to be observed in path j than path k. We utilize the correlation of appearance between different streams in order to better estimate *bestscore*.

Given an object p we define the random variable $X_i \in \{0, 1\}$ such that $X_i = 1$ if p has been observed in stream s_i in the current window. Similarly $X_i = 0$ if p has not been seen in stream s_i. We are interested in calculating the conditional probability of observing p in s_i if p has been already seen in stream s_j: $Pr(X_i = 1|X_j = 1)$. In the following we first show how this information is utilized and then present how it can be computed efficiently.

Given $I = \{i_1, i_2, ..i_r\} \subseteq \{1, .., d\}$ for an object p which indicates the streams where p has been observed in, for each $j \notin I$ assume we know the conditional probability of observing p in s_j : $Pr(X_j = 1|X_k = 1; k \in I)$. As before, $p^l.\widetilde{bestscore} = f(\tilde{v}_b(1), ..., \tilde{v}_b(d))$ where $\tilde{v}_b(y)$ is estimated as follows:

$$\tilde{v}_b(y) = \begin{cases} p.value(y) & y \in I \wedge p.t_y > p.t_{i_j} \\ 0 & y \in I \wedge p.t_y < p.t_{i_j} \\ Pr(X_y = 1|X_k = 1; k \in I) & otherwise \end{cases}$$

$\tilde{v}_b(y)$ is naturally smaller than or equal to $v_b(y)$. Since the aggregation function f is monotonically increasing $p^l.\widetilde{bestscore}$ in a lot of cases will be much smaller than $p^l.bestscore$. As a result of this, the interval size would be smaller. Decreasing the interval size results in a smaller dominance set. At the same time, since we are estimating *bestscore* realistically the quality of results should remain high.

We now describe how the required correlation statistics can be computed. Since $Pr(X_i = 1|X_j = 1) = \frac{Pr(X_i=1 \wedge X_j=1)}{Pr(X_j=1)}$, $Pr(X_i = 1|X_j = 1)$ can be statistically estimated as the join size of s_i and s_j, $|s_i \bowtie s_j|$ weighted by $1/|s_j|$ where $|s_j|$ is the number of distinct elements observed in a window in stream s_j. This is because each object appears in each stream at most once during an active window. This can be generalized to the case where an object is seen in several streams and we are interested to know with what probability it will appear in

the rest of the streams.

We use the FM sketches to estimate these probabilities. FM sketches were first proposed in [FM85] to probabilistically estimate the cardinality of a multi-set M. These hash sketches use a pseudo-uniform hash function $h() : M \rightarrow [0, 1, \ldots, 2^L)$. In [DF03], Durand and Flajolet presented a similar algorithm (*super-LogLog counting*) which reduces the space complexity and relaxes the required statistical properties of the hash function.

Briefly, hash sketches work as follows. Let $\rho(y) : [0, 2^L) \rightarrow [0, L)$ be the position of the least significant (leftmost) 1-bit in the binary representation of y; that is,

$$\rho(y) = \min_{k \geq 0} bit(y, k) \neq 0,\ y > 0$$

and $\rho(0) = L$. $bit(y, k)$ denotes the k-th bit in the binary representation of y (bit-position 0 corresponds to the least significant bit). In order to estimate the number n of distinct elements in a multi-set S we apply $\rho(h(d))$ to all $d \in S$ and record the least-significant 1-bits in a bitmap vector $B[0 \ldots L-1]$. Since $h()$ distributes values uniformly over $[0, 2^L)$, it follows that $P(\rho(h(d)) = k) = 2^{-k-1}$.

Thus, when counting elements in an n-item multi-set, $B[0]$ will be set to 1 approximately $\frac{n}{2}$ times, $B[1]$ approximately $\frac{n}{4}$ times, etc. Hence, the quantity $R(S) = max_{d \in S}\rho(d)$ provides an estimation of the value of $\log_2 n$. Techniques which provably reduce the statistical estimation error typically rely on employing multiple bitmaps for each hash sketch, instead of only one. The overall estimation then is an averaging over the individual estimations produced using each bitmap.

In our case we are interested in the cardinality of the intersection of two streams s_i and s_j in a window. Since we know that the number of valid distinct elements in s_i is equal to the number of elements arriving in a window, we can use the fact that $|s_i \bowtie s_j| = |s_i \cup s_j| - (|s_i| + |s_j|)$ to estimate $|s_i \bowtie s_j|$. If we keep an FM sketch for each of s_i and s_j, the union of these two sketches can be used to estimate $|s_i \cup s_j|$. Since $(|s_i| + |s_j|)$ is known, it is then possible to estimate $|s_i \bowtie s_j|$. Note that we can similarly estimate $|s_i \bowtie s_j \bowtie s_k|$ or the cardinality of higher number of joins by only keeping FM sketches per stream.

We still need to take care of the sliding window factor: if an object o is seen in s_i at time t_i and in s_j at time t_j and t_i and t_j do not belong to the same window, o shouldn't appear in $s_i \bowtie s_j$. As mentioned in [DGIM02], the FM sketches can be adapted to estimate the number of distinct elements in a sliding window by associating a bitmap of size $O(logW)$ with each of the bits in the sketch. Whenever a bit is (re)set by an object in the stream, its associated time-stamp is updated to that of the object. In this way when evaluating the number of distinct elements in the current window, only those bits which could have been set in the current window are considered. This increases the storage requirement of the FM sketch with a logarithmic factor.

We note that estimating bestscore can be further tuned to provide more accurate results at the expense of consuming more space. In the above computations, we use the probability of appearance for estimating bestscores, however,

this can be refined to using different probabilities based on the specific values observed. Depending on the application and the desired level of accuracy, such refinements can be applied to our algorithm which come with the cost of maintaining the necessary statistics, but result in more accurate results.

4.4 Multiple Occurrences

So far, in all our described algorithms we assumed that each object is observed in each stream at most once. In some real world scenarios, such as the flow example, this assumption may not hold. Here, we shortly describe how each algorithm is extendable to the case where re-occurrences happen. In SLA, if an object reoccurs in a stream, its previously observed values are updated with aggregating the old seen value with the new one, and the newly seen value is also inserted in the corresponding sorted list. When performing the TA algorithm, a lookup in each list may return several instances, in such a case the largest value is used. When a tuple expires, its corresponding item in the sorted list is also removes. Assume the maximum number of re-occurrences in a stream is n. The currentscore and bestscore of an instance object are updated with regard to the number of times this object has been observed in a stream as well as in which streams it has been observed. In particular we use $n - n_i$ as the maximum value if an object has been observed n_i times in stream i (assuming *sum* for aggregating multiple occurrences). Also, similar to SLA, the values of existing observations are updated by aggregating a newly seen value and a new instance object is produced. All calculations are straight forward by keeping the number of times an object has been observed. In our approximate algorithm, instead of using n, we estimate the maximum number of re-occurrences of an object by measuring the self join size of a stream using one of the existing approaches such as AMS sketches [AGMS02].

4.5 Fixed Memory

While the previous approximate algorithm prunes more objects and consumes less memory, it can not be directly used in scenarios where the amount of memory is *strictly* fixed. In this Section we consider top-k monitoring over multiple non-synchronized streams when the amount of available memory is fixed. This problem is similar to the famous *paging* problem, where it is decided which cache entry is replaced by a new incoming item. We propose an optimum offline algorithm for this problem. The offline algorithm (OPT) is designed under the assumption that all future tuples are known in advance to the algorithm. Therefore it gives the best achievable precision under fixed memory.

Consider a fixed memory size of m. For the sake of analysis, assume we have streams of finite known length. A new tuple arrives in each stream at each timestamp and we report the top-k items every timestamp. Without loss of generality we assume that first, the top-k evaluation is performed and then

the new incoming tuple is processed. We use the same instance creation as explained in Section 4.2.2.

If the memory is not full when a tuple arrives, we update the corresponding existing object instances (if any), create a new instance and place it directly in the memory. Otherwise the newly created object instance should be dropped or it should replace an existing instance in the memory. This is similar to the *paging* problem, however the cost of our problem is different. In paging, the cost is defined by the number of page faults. In our problem however, the cost is defined by the reported set of top-k objects against the true top-k results, every timestamps. Each time we drop an instance which in the ideal case of unlimited memory would have been reported as a top-k item later, our algorithm incurs a cost. Since we are now focusing on precision, the cost is equal to the number of times this tuple would appear in the true top-k set in future. Note that unlike paging, in our case once an instance is evicted it is never placed in the memory again.

OPT is assumed to know the future tuples in advance. Therefore it knows the set of future true top-k objects and based on this information it decided which object should be evicted from the memory among the $m + 1$ valid instances.

The above optimization problem can be formulated as a network flow problem. We describe how such a flow graph should be constructed. Our ideas are similar to the k-server problem (generalization of the paging problem) and also solutions for approximate join processing [DGR03].

The source node s has a supply m (size of our memory) and the sink node t has demand m. The rest of the nodes do not have any supply or demand. Apart from the source and sink nodes, each node in the graph corresponds to an instance residing in the memory at a certain time. We assume at each timestamp only one tuple arrives, so at each timestamp one new instance is created. For simplicity we denote an instance which was created at time i with ν_i. Note that this is regardless of the object this instance presents, ν_i and ν_j are two different instances created at time i and j which may represent the same object or don't. A node with label $\nu_i : j$ means that instance ν_i which was created at time i is in memory at time j. Clearly $i \leq j$, since an instance can not be place in memory before it is created. Edges among nodes model possible combinations of keeping or replacing an instance. All edges have capacity 1, which means they can transmit any value in $[0, 1]$. Edges are placed between nodes with consecutive times: we connect any two nodes $\nu_i : j$ and $\nu_i : j + 1$. A flow on such an edge shows that ν_i is still in memory at time j and was not evicted before. We also place edges between $\nu_i : j + 1$ and $\nu_{j+1} : j + 1$. Such edges represent replacing tuple ν_i with ν_{j+1} at time $j + 1$. Finally we connect the first m incoming tuples to the source and connect all nodes which have a time component larger than $|S| - m$ to the sink.

Costs are assigned to edges in such a way that an optimal flow corresponds to maximizing recall. We show the set of true top-k objects instances at time j with R_j. We assign cost factor -1 to $\nu_i : j - 1 \rightarrow \nu_i : j$ if $\nu_i \in R_j$. All other edges are assigned a cost factor of 0.

Solving a *min-cost* linear flow problem corresponds to the optimal solution for our problem. With the above construction each node will receive at most a flow one. The edges do not allow a flow more than m in the graph which corresponds to our memory constraint. Also the edges are placed in such a way that if an instance is replaced by some other instance it does not have a chance of being back in the memory later. Costs are assigned such that only instances in the true top-k can contribute to minimizing the cost flow. Also we should ensure that all flows are either 0 or 1 : if we have flow 0.4 on edge $\nu_i : j \rightarrow \nu_i : j+1$ and flow 0.6 on $\nu_i : j \rightarrow \nu_j + 1 : j+1$ this means that 40% of tuple ν_i and 60% of tuple ν_{j+1} where in memory at time $j+1$ which is not acceptable. This is ensured by a theorem from network flows: if all costs and flows of the graph are integral an optimal integral solution also exists [Roc98].

To emphasize the hardness of this problem, we show that for arbitrary streams, the *competitive* ratio of any online algorithm is proportional to the window size W which can be arbitrarily large.

Theorem 2 *For a fixed memory of size m and a sliding window of size W, the competitive ratio of any online algorithm is larger than $W - m$.*

For the proof see the appendix. We do not discuss possible online solutions to this problem. Similar to the case for approximate joins over streams [SW04], for different stream models different online algorithms may be appropriate. More discussion on this is beyond the scope of this thesis.

4.6 Experiments

We have implemented a light-weight stream processing engine in Java 1.6 providing standard operators (project, select, join), data source wrappers that replay existing benchmark data, and data generators for synthetic data. The real world dataset is stored in an Oracle 11g database, the synthetic data is created on the fly.

4.6.1 Setup

Datasets

Synthetic Dataset: To obtain a better understanding of the impact of data characteristics on the performance and accuracy behavior of the algorithms under comparison, we first start by employing a data generator to produce streams with tunable inter-stream correlation. We use the following methodology: to generate n streams of each distinct items but with controlled correlation, the data generator keeps track of each item sent in one of the n streams separately. In addition, it keeps a sliding window of the last C values for each stream. To produce a new item to be inserted in a stream, the generator picks with probability ξ a value that is currently in the sliding windows of the other streams, and with probability $(1 - \xi)$ draws a fresh item from a random number generator, ignoring those items that have already been sent. The stream is not

materialized in a database (or file); it is created on demand.

Real World Dataset: We use the WorldCup98 [1] dataset as our real dataset. This dataset consists of requests made to the 1998 World Cup Web site between April 30 and July 26, 1998. During this period the site received 1,352,804,107 requests. Each tuple consists of several fields. Each tuple has a *clientID* field which uniquely identifies the client which issued the request. The *server* field indicates which server handled the request and specifies the stream ID for us. *Size* shows the number of bytes in the response. The query asks for the clients who have downloaded the highest number of bytes in a given time frame. The data is stored in an Oracle database with a B+ tree index on (server, time ASC, client, bytes). The stream per server consists of multiple requests for each user (e.g., each HTTP request). We have implemented a stream operator that pre-aggregates these fine grained events in the following way. As is commonly done in this scenario, the preprocessing operator pre-aggregates chunks of a certain time period (δ_t) and sends the pre-aggregated values to the consuming stream operator. For each stream, client ids occurring in the same pre-aggregated stream (streams of pre-aggregated chunks) multiple times will be broken in different clients. This reflects in particular changes of IP addresses. These chunked streams are then processed by our top-k operators.

Algorithms under Comparison

We evaluate the following proposed algorithms:

SLA: This is the algorithm based on multiple sorted lists as explained in Section 4.2.1. It provides the baseline for assessing the quality of results as explained later.

EAA: Our proposed algorithm as described in Section 4.2.2. It uses the interval dominance check to discard instances which are not part of the k dominance set. Similar to SLA, EAA provides exact results and retains the least number of objects which can guarantee this.

approxAlgo: This is the approximate algorithm as proposed in Section 4.3. It estimates the best possible score of each instance more realistically based on stream inter correlations.

Measures of Interest

Memory Consumption: The number of retained objects is the dominant factor in storage. FM sketches generally consume very little space which is constant for each stream and negligible compared to the number of data points which should be stored. We report on the number of items retained for each of the algorithms under comparison as a measure of storage. Items are tuples

[1] http://ita.ee.lbl.gov/html/contrib/WorldCup.html

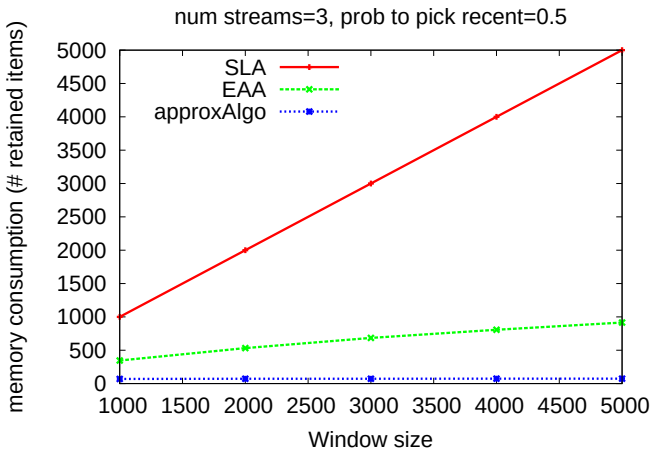


Figure 4.6: synthetic dataset: Memory consumption when varying the window size W

in case of SLA, as we keep tuples separately without aggregating them. We ignore the k_{max} materialized aggregated results, as $k_{max} << W$. For EAA and approxAlgo, an item is an aggregated object instance.

Precision: We report on the precision, i.e., the number of relevant data points among returned top-k results as the effectiveness metric. The relevance is defined by the SLA method which keeps all valid tuples. Assume SLA reports A as the set of top-k results, this set is the ground truth. So if set B is returned as the top-k in another algorithm, this algorithm's precision is calculated as: $precision = \frac{|A \cap B|}{k}$.

Relative Error: Since one of our proposed algorithms provides approximate results, we are interested in measuring the quality of the approximate results: how far from the true result is a returned element. Assume A is the ground truth set and B is the set of top-k results returned. The rank i element in set X is denoted by x_i. Then the relative error is calculated as: $\frac{1}{k} \sum_{i=1}^{k} |a_i - b_i|$.

All reported measurements are averaged over 50 evaluations.

4.6.2 Experimental Results

Synthetic Dataset

In this Section we describe the experimental results for the synthetic dataset. In the first set of experiments we investigate the effect of the sliding window size

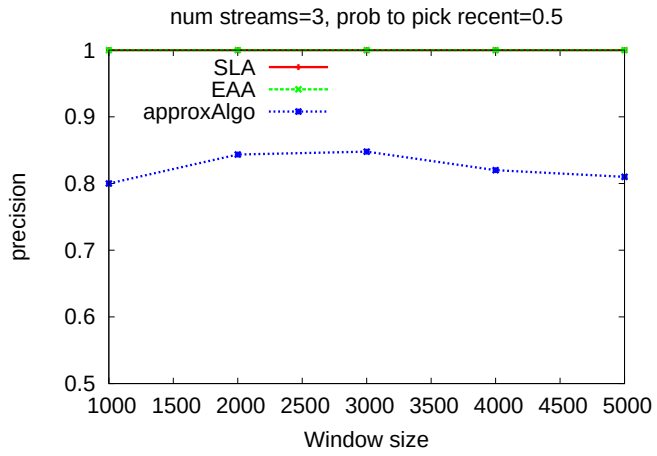


Figure 4.7: synthetic dataset: Precision when varying the window size W

on memory consumption and precision. SLA keeps all valid tuples, therefore has a memory consumption equal to size of the sliding window. EAA decreases this by dropping objects which will never be part of the top-k. EAA's memory consumption is therefore the minimum required to guarantee exact results. Figure 4.6 shows the results of this experiment. As expected, EAA has smaller memory consumption compared to SLA. *approxAlgo* reduces the memory significantly and shows very little increase as opposed to EAA which has a linear growth with the window size. Figure 4.7 presents the achieved precision in this case. EAA has precision 1 as it returns exact results. *approxAlgo* shows very small variations in precision (between 0.80 and 0.84) as the window size changes, indicating its robustness.

In Figures 4.8 and 4.9 we observe the effect of varying the correlation parameter between streams. The number of streams is fixed to 2 in this case and we show the results for different algorithms for window sizes 500 and 1000. SLA has memory consumption equal to the sliding window. As the correlation between streams increases (which is the result of increasing ξ) the memory consumption for EAA decreases. This is in accordance with our results from Lemma 1: by increasing ξ the average score interval decreases, as more objects are seen in all streams. *approxAlgo*'s memory consumption shows small decrease (from 130 to 84 for W=1000 and from 115 to 75 for W=500). Figure 4.9 reports the precision when changing ξ. EAA and SLA have precision 1. For *approxAlgo*, precision values increase as ξ increases, which is due to decreasing the uncertainty. However, the variations between precision values are small for both window sizes indicating the good quality of our correlation estimation technique.

We present the results of changing the number of streams, which corresponds

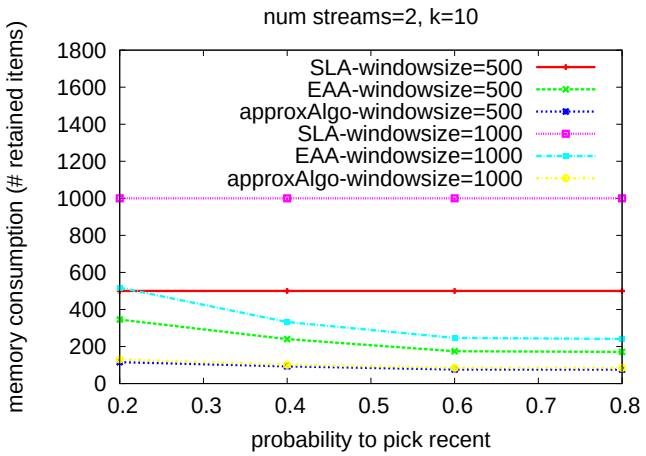


Figure 4.8: synthetic dataset: Memory consumption when varying the probability to pick recent

streams	sizeEAA	sizeApprox	precApprox	errorApprox
3	345	69	0.80	0.0174
4	450	92	0.81	0.0242
6	495	115	0.76	0.0408

Table 4.1: Results when changing number of streams for the synthetic dataset. Windowsize =1000, k=10 and ξ =0.5

to dimensionality of the objects monitored in Table 4.1. We do not show the results for SLA, as its memory consumption is fixed (1000 which is equal to the window size). The memory consumption increases with increasing dimensionality for both EAA and *approxAlgo*. Again this is due to the increase in interval size. Precision and relative error are shown for *approxAlgo*. Note that these values are respectively 1 and 0 for EAA.

Table 4.2 reports the effect of parameter k. Increasing k clearly increases the storage as the number of elements which are not dominated by more than k elements naturally increases. This is apparent for both EAA and *approxAlgo*. We observe better precision for bigger values of k for *approxAlgo*: equal number of missing objects from the top-k results has less effect for larger values of k. Relative error shows the same trend (decreases as k increases).

Worldcup Dataset

Table 4.3 reports on the average performance for the worldcup dataset when varying the number of streams which represents the number of servers in this scenario. Similar to the synthetic dataset described above, our approximate algorithm (*algoApprox*) causes drastic performance gains in memory consumption

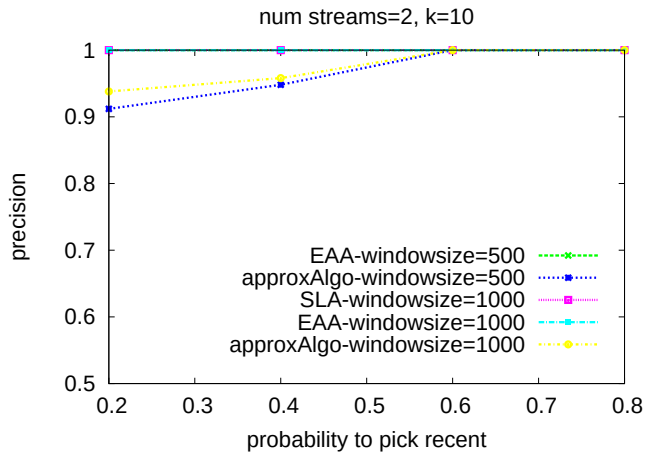


Figure 4.9: synthetic dataset: Precision when varying the probability to pick recent

k	sizeEAA	sizeApprox	precApprox	errorApprox
10	345.3	69.26	0.80	0.01743
20	396.3	105.6	0.86	0.0103
50	450.0	198.7	0.96	0.0027
100	481.7	315.1	0.99	0.0009

Table 4.2: Results when changing k for the synthetic dataset. Windowsize =1000, numstr=3 and ξ=0.5

with only minor losses in result accuracy, yielding a precision between 0.92 and 0.95.

Similarly, Table 4.4 reports on the performance and accuracy numbers when changing the window size W, showing major decrease in memory consumption with minor losses in accuracy.

Given the above results, we observe the effectiveness of the proposed algorithms. The pruning of dominated items in our EAA algorithm successfully decreases the memory consumption while still guaranteeing exactness of the result set. It gives the basis for the optimization steps introduced by our approximate algorithm, that further decreases memory consumption. The penalties in result quality is almost negligible given the achieved reduction of items to keep in memory. The insights learned, in particular from running experiments using the more controllable synthetic dataset, is that the smaller the correlation of the involved streams the higher the impact of our approximate method.

streams	sizeEAA	sizeApprox	precApprox	errorApprox
2	217.46	80.1	0.92	0.0113
3	250.0	110.9	0.93	0.0136
4	250.0	139.8	0.95	0.0097

Table 4.3: Results when changing number of streams for the Worldcup dataset. Window = 500 and δ_t=5000ms and k=20

W	sizeEAA	sizeApprox	precApprox	errorApprox
500	217.46	80.1	0.923	0.0113
750	310.66	80.9	0.959	0.0082
1000	407.09	80.86	0.974	0.0084

Table 4.4: Results when changing the window size W for the Worldcup dataset. number of streams = 2 and δ_t=5000ms and k=20

4.7 Conclusion

We address the problem of processing continuous top-k queries over multiple non-synchronized data streams where exact score computation is seldom possible. To the best of our knowledge, this problem has not been addressed before in the literature. We start by a describing a solution which is a direct adaptation of the TA algorithm, called SLA which serves as our baseline method. Given that SLA requires all incoming tuples to be stored and this is infeasible in many streaming scenarios with limited resources, we aim at storing only those tuples which are necessary for providing exact results. We extend the notion of *dominance*, proposing an early aggregation scheme which enables efficient pruning of objects. However, we theoretically show and experimentally confirm, that the necessary number of elements which have to be kept to guarantee exact results, grows linearly with the window size. Our approximate approach is based on the observation that the size of dominance set is a direct factor of the difference between best and current scores. We leverage the correlation appearance between different streams, which is usual in real world scenarios, to estimate bestscore in a less optimistic way than considering the best possible scores for unseen attributes. As seen in the experiments, this method provides highly accurate results while reducing the number of retained objects dramatically.

Chapter 5

Information Filtering in Web 2.0 Streams

5.1 Introduction

The world has turned into one large-scale interconnected information system with millions of users. With the advent of Web 2.0, yesterday's end users are now content generators themselves and actively contribute to the Web. Each user action, for example uploading a picture, tagging a video, or commenting on a blog, can be interpreted as an event in a corresponding stream. Given the immense volume of this data and its vast diversity, there is a vital need for effective filtering methods which allow users to efficiently follow personally interesting information and stay tuned.

Currently popular methods place the filter on the data sources: mechanisms such as RSS and atom are used to notify users of newly published data on their favored weblogs or news portals. However, with the currently available functionalities, users can only decide to be notified of new posts on certain blogs or follow certain other users as in Twitter [1]. This limits the number of subscriptions users make, as otherwise the amount of received information will be overwhelming for human processing. On the other hand, traditional information filtering systems [BM96, YGM94a], aggregate all available information sources and allow users to specify their interests as profiles. Given a similarity measure between the data and the profiles, only data which passes a certain quality-based threshold is returned to the user. Although this diversifies the returned results as opposed to the previous method, it can easily result in flooding the user with returning too many data. Choosing a suitable threshold to avoid overwhelming the user or returning very few results is very hard due to the ever changing nature of incoming data. This calls for a system which deems relevance as *relative* to the existing pool of information [MP09], as opposed to *absolute* relevance. Furthermore, to account for the desire of consuming new information and to prohibit repeatedly

[1] http://www.twitter.com

returning highly relevant, but old information, data is considered valid for only a certain time interval, controlled by a sliding window. Note that in the context of Web 2.0, all information come with explicit temporal annotations e.g., *written at*, *uploaded at*, which makes them natural items of a temporal stream; therefore the definition of a sliding window is meaningful. The dynamism introduced as a result of considering relevance relative in a frequently changing information pool, as well as the scale of our envisioned system in handling huge number of users, poses fundamental new challenges which were nonexisting previously.

As an illustration, emphasizing the importance of the addressed problem, consider a small scale case where $100,000$ profiles are maintained at a single server. The naive approach consists of evaluating every profile against the incoming documents and re-evaluating the profiles upon result expiration from scratch. According to our experiments, these operations, disregarding the cost for indexing the documents, i.e., removing stopwords, calculating TF/IDF values take on average, orders of tens of milliseconds per document on a quad-core Intel Xeon CPU E5530 @2.4GHz machine. This means that the maximum supported rate of incoming documents would be in order of hundred documents per second which is relatively small, given that today only in Twitter, around 600 messages are produced per second [2].

5.1.1 Problem Statement

We consider a stream $\mathcal{S}$ of documents where each document is uniquely identified and consists of a weight vector, as in classical Vector Space Model, as well as its arrival time: $\mathbf{d} = \langle id, time, d \rangle$. Assuming m distinct terms available for content identification, d is an m-dimensional vector $d = (w_1, ..., w_m)$, where w_i is the weight assigned to the i-th term. Terms which do not appear in the document have a zero weight. Any of the usual scoring schemes such as the TF/IDF methodology can be used for assigning the weights. We further assume *in-order* streams; items arrive in the same order that they are generated. In most streaming scenarios, as well as ours, recent items are of more interest than old ones. This is captured by the sliding window model. A sliding window (W) is assumed over the stream and items are considered *valid* while they belong to this window. Sliding windows can be either count or time based, i.e., bounding the number of items either by count or focusing only on those that occurred in a particular time interval. Our solution can be applied to both types.

Similar to web search, we assume user interests, called *profiles*, are expressed as sets of terms with corresponding weights: We denote a profile by $p = (u_1, ..., u_m)$, where u_i specifies the importance of the i-th term to the user. The relevance of a document to a profile is determined by a scoring function: $sim(d, p) = g(f_{w_1}(u_1), ...f_{w_m}(u_m))$. We make the following assumptions regarding g and f:

- **Monotonicity**: We assume g and f_i's are monotone. $g(x_1, ..., x_m)$ is

[2] http://blog.twitter.com/2010/02/measuring-tweets.html

monotone if $g(x_1, ..., x_m) \leq g(x'_1, ..., x'_m)$ whenever $x_i \leq x'_i$ for all i and similarly for f_is.

- **Homogeneity**: We assume g and f_i's are homogeneous, i.e., they preserve the scalar multiplication operation: $g(ax_1, ..., ax_m) = a^r g(x_1, ..., x_m)$. For simplicity we assume that the homogeneity degree is 1: $r = 1$. However, our solution can be applied for higher degrees of homogeneity as well.

Note that taking the f functions as multiplication and g as summation, we arrive at the widely used cosine measure as the scoring function for normalized vectors.

We consider a main memory model and treat each profile as a top-k query. All queries should be continuously monitored to keep the users up-to-date while the valid pool of information changes due to arrival of new documents or expiration of old ones. This goal involves two main tasks: first is efficient and scalable profile filtering in order to avoid comparing an incoming document against the large set of *all* existing profiles. Second is maintaining the top-k results of each profile as the window slides and some documents become invalid. In both tasks we focus on efficiency which is a necessary step towards ensuring scalability.

This Chapter is based on our work in [HMA10] and is organized as follows: Section 5.2 briefly describes the general structure that we consider together with a baseline algorithm. Section 5.3 describes an efficient algorithm for profile filtering. Section 5.4 discusses the skyline-based algorithm for result maintenance which aims at avoiding re-evaluations to high extent. Section 5.5 presents the experimental evaluation and Section 5.6 concludes this Chapter.

5.2 System Model and Structure

In this section we briefly describe the general structure that we consider. As mentioned in Section 5.1.1 we consider one data stream as the input to our system. We aim at providing real-time continuous exact results to the users, therefore results returned as the top-k most relevant documents for each profile should consist of the top-k results over *all* valid documents in the system at each instance of time. With small number of queries (profiles) or infrequent updates in the result set of each profile, all queries could be re-evaluated at certain times to this end. However, given large number of profiles, this solution does not scale to provide real-time monitoring of all profiles' results. Instead, we index the profiles and assess the suitability of a new document for each profile as the document arrives in the system. We avoid evaluating all profiles against a new incoming documents, by a *profile filtering component*. The profile filtering component receives a newly arrived document as input and returns a set of profiles which should be updated with regard to this document. As the baseline, we maintain an inverted list structure in the profile filtering: For each term t we keep a list of profiles which contain this term, i.e., weight of t is larger than zero for those profiles. We also maintain a hashtable of profiles,

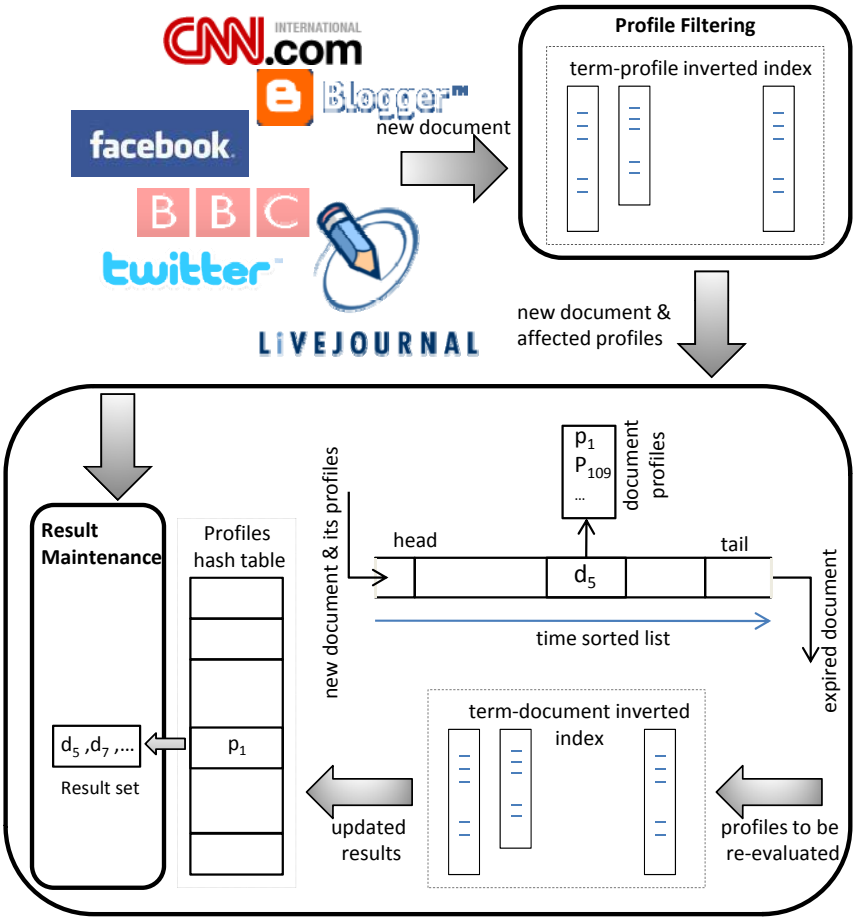


Figure 5.1: The Overall Structure

where profiles with a pointer to their current result set are stored. When a new document d arrives, we evaluate all entries of all inverted lists corresponding to a non-zero weighted term in d. Evaluating an entry corresponding to a profile p consists of calculating $sim(d, p)$ and comparing this with the current rank k result of p which can be known by accessing the profile hashtable. If the new document has a better similarity score, p's result set is updated with it.

As the window slides, some documents expire and the set of valid documents changes. Some profiles may need to be re-evaluated as the expired documents were part of their result set. The *result maintenance component* is concerned with efficiently performing this task. We keep a simple time-sorted list for tracking valid documents: newly arrived documents are inserted at the head and those which expire drop out from the tail. For each document we maintain the set of profiles to which this document is a top-k result. Therefore, when a document expires, the set of profiles which should be re-evaluated is known. The actual method for performing the re-evaluation is not a main concern of this Chapter. However, we assume sorted inverted lists are maintained such that the usual threshold algorithm (TA) [Fag02] is employed for query evaluation. Figure 5.1 presents the general components and structures we consider.

5.3 Efficient Profile Filtering

Similar to the baseline described in Section 5.2, we use an inverted index of profiles to avoid examining all the existing profiles against the newly arriving documents. In contrast to the former approach, we utilize sorted versions of these lists. As we will describe in this Section, processing these sorted lists will enable early stopping to further reduce the number of examined profiles. Our method is similar to the well-known TA (threshold algorithm) [Fag02] which is widely used in information retrieval.

As previously mentioned, we consider each profile as a continuous top-k query over the stream of incoming documents. Let $p.s$ denote the similarity score of the ranked k document with regard to profile p. Let $T = \{t_1, t_2, ..., t_m\}$ be the set of distinct terms considered for content identification of both profiles and documents. $p.u_i$ represents the corresponding weight assigned to term t_i in p. For each term t_i, we build a list l_i containing $\langle p.id, p.v_i \rangle$ pairs where $p.v_i = p.u_i/p.s$ and the list is sorted in decreasing order based on v_i values. $p.id$ denotes the unique identifier of profile p. A profile with l terms will only appear in l of such posting lists.

Assume a document d with the feature vector $d = (w_1, ...w_m)$ arrives in the system. We do sorted accesses in a round robin fashion to all posting lists l_i where $w_i > 0$. When a profile p is seen under one of these lists, we access the profile hash table for its complete weight vector and consequently calculate the similarity score between d and p. The result set of profile p is updated if

$$sim(d, p) > p.s$$

where as mentioned before $sim(d, p)$ satisfies the two conditions described in

Section 5.1.1.

While accessing the sorted lists in a round robin fashion, we also check the following stopping condition. For a list l_i let $\underline{v_i}$ be the last observed value under sorted access. We stop the above procedure when $g(f_{w_1}(\underline{v_1}), ...f_{w_m}(\underline{v_m})) < 1$.

We call the above procedure *COL-filter* (Completely Ordered Lists) and show that it is complete: all profiles for which a new incoming document serves as a top-k result are identified before the stopping condition.

Theorem 3 COL-filter *identifies all profiles for which a new incoming document d is a top-k result.*

As g is monotone and the lists are sorted, reaching the stopping condition means that for any non processed profile p, $sim(p, d) < p.s$. For a detailed proof please see the appendix.

Since the number of lists which are processed could be much larger than the number of terms a profile has, we take into account the maximum number of terms a profile can have in calculating the stopping condition. Assume the maximum number of terms per profile is m'. The stopping condition could be checked per list, i.e., the procedure can stop processing one list while the other lists should still undergo the procedure. Let I be a subset of size m' of the lists under process. We define the following:

$$f^I_{w_i}(\underline{v_i}) = \begin{cases} f_{w_i}(\underline{v_i}) & \text{if } l_i \in I \\ 0 & \text{otherwise} \end{cases}$$

Also, let $\mathbb{I}_j$ be the set of all subsets I, where I includes l_j. The algorithm stops processing l_j if

$$max_{I \in \mathbb{I}_j} g(f^I_{w_1}(\underline{v_1}), ...f^I_{w_m}(\underline{v_m})) < 1$$

If g is symmetric, i.e., its value at any m-tuple of arguments is the same as its value at any permutation of that m-tuple, it is enough to do the test for one set I_{max} which contains the $(m' - 1)$ lists with the largest $f_{w_i}(\underline{v_i})$ values along with l_j. The general steps are shown in Algorithm 3.

While COL-filter enables early stopping and avoids accessing and assessing all profiles which appear in an inverted list of a term in an incoming document, it incurs high costs for maintaining the inverted lists. Contrary to standard information retrieval inverted lists, where the lists are static and change rarely, the sorted lists in COL-filter change frequently. This is because the values we use for sorting depend on the similarity scores of profiles which change with time, as new documents arrive or old ones expire and are removed from the system. Note that each time a profile p is updated, i.e., a new incoming document qualifies as its top-k result, $p.s$ changes. As a result $p.v_i = p.u_i/p.s$ changes, therefore p's corresponding tuples in all lists which p appeared in should be updated. As a consequence of the high dynamism inherent in the system, which is due to the high rate of incoming documents and their expiration, the number of

```
ProfileFilter Input: d = (w_1, ..., w_m)
toProcess = ∅;
toUpdate = ∅;
if w_i > 0 then
  | toProcess.add(l_i);
end
while toProcess ≠ ∅ do
  | foreach list l_i ∈ toProcess do
  |   | p = l_i.getNext().getProfile();
  |   | v_i = p.v_i;
  |   | p.s = p.getScore(k);
  |   | if sim(p, d) > p.s then
  |   |   | toUpdate.add(p);
  |   | end
  | end
  | foreach list l_j ∈ toProcess do
  |   | if max_{I∈𝕀_j} g(f^I_{w_1}(v_1), ...f^I_{w_m}(v_m)) < 1 then
  |   |   | toProcess.remove(l_j);
  |   | end
  |   | if !l_j.hasNext() then
  |   |   | toProcess.remove(l_j);
  |   | end
  | end
end
return toUpdate;
```

Algorithm 3: The COL filtering algorithm

necessary updates can be very high. The cost of maintaining the lists sorted can therefore overshadow the benefits of early stops. In the following, we propose a relaxation to completely sorting the lists, which requires significantly fewer number of updates and is cheaper to maintain.

5.3.1 Partially Ordered Lists

We aim at decreasing the cost of maintaining the sorted lists by grouping entries and ordering the entries only based on a fixed number of predefined boundaries instead of maintaining full order. These boundaries are then used to test the stopping condition. Our *Partially Ordered List* method, *POL-filter*, is described below.

Similar to COL-filter, we maintain inverted lists for each term t_i, denoted by l_i with entries $\langle p.id, p.v_i \rangle$ as defined above. For each list l_i we consider r groups which we identify by their boundaries: $b_{i1} > ... > b_{ir}$. The entries in l_i are grouped based on these boundaries. An entry $\langle p.id, p.v_i \rangle$ belongs to the group b_{ij} if $p.v_i \geq b_{ij}$ and $p.v_i < b_{i(j-1)}$, where the second condition is assessed only for $j > 1$. The entries inside one group are not kept sorted. To process an

incoming document $d = (w_1, ...w_m)$, we start with the first group b_{i0} in all lists l_i where $w_i > 0$ and calculate the similarity scores of profiles in these groups with regard to d. The complete weight vector of a profile can be known, when necessary, by accessing the profile hash table in constant time. A profile's result set is updated with d when $sim(d, p) > p.s$. We continue to assess the profiles in the next group in each list only if the following stopping condition is not satisfied:

$$g(f_{w_1}(b_{10}), ...f_{w_m}(b_{m0})) < 1$$

In the following rounds, the algorithm reads all profiles which appear in group $b_{j(t+1)}$ where j identifies the list with the largest $w_j b_{j(t)}$ value and t is the last group assessed in list j. The stopping condition, replacing $b_{j(t)}$ with $b_{j(t+1)}$ in the above equation, is assessed each time a new group $b_{j(t+1)}$ is processed. It is easy to see, similar to the proof of Theorem 3, that this procedure is complete. Note that similar to COL-filter we can use the fact that the number of terms per profile is much smaller than the number of lists which are under process and improve the stopping condition.

The update cost in POL-filter is limited to maintaining the groups in each list. Since the entries in a group are not sorted, a hash table which provides constant insert and removal costs could be used to add or remove entries to groups. We move a profile p from a group when $p.s$ changes *and* the group membership does not hold anymore for the current group. In this case, the algorithm identifies the newly qualified group and the two affected groups are updated. Note that $p.s$ can change due to arrival of a new document which qualifies as p's top-k result or expiration of a previously top-k document.

5.3.2 Boundary Selection

While POL-filter decreases the maintenance cost, its effectiveness on early stopping depends on the selected group boundaries. If the b_i values are chosen to be too big, the stopping condition is not satisfied and the algorithm processes all the profiles in a list. On the other hand if they are chosen to be too small, the algorithm may process too many unnecessary profiles before it stops. Fixing the number of groups to r, we measure the extra cost POL-filter incurs by processing unnecessary profiles and aim at minimizing it. To calculate this extra cost, we assume that the lists are sorted also inside the groups, and calculate the number of extra profiles processed in a group. For simplicity let us assume we have a single term t and its corresponding list l. We later show how our discussion is extended to multiple lists. We denote the weight of t in an incoming document with w and the entries in l by $p.v$. We consider r groups with boundaries $b_1 > ... > b_r$. In case of a single list, if group b_i's profiles have been assessed, the POL-filter's stopping condition is $g(f_w(b_i)) < 1$. Let W and V denote the random variables corresponding to w, weight of t, and v values. Assume $f_W(x)$ and $f_V(x)$ respectively show the probability distribution functions of W and V. Also assume the size of list l is q and the number of documents with term t is n. The following is the cost which POL-filter incurs:

$$\sum_{i=1}^{r} n \int_{b_i}^{b_{i-1}} f_V(x) \int_0^1 f_W(z/x) qPr(b_i < V < x) \delta z \delta x$$

where b_0 is the largest value V can have and b_r the smallest value. Since we have assumed the profiles are sorted also in the groups, $n \int_{b_i}^{b_{i-1}} f_V(x) \int_{\infty}^{1} f_W(z/x) \delta z$ counts the number of times the stopping condition is satisfied for a profile in group b_{i-1} and $qPr(b_i < V < x)$ counts the number of extra profiles POL-filter reads in this case. Since POL-filter checks the stopping condition each time a new group is assessed, it reads at most "*size of a group*" extra profiles compared to COL-filter. The above simplifies to:

$$\int_{b_0}^{b_r} nqx f_V(x) Pr(W < 1/x) Pr(x) \delta x$$

$$-\sum_{i=1}^{r-1} Pr(b_i) \int_{b_i}^{b_{i-1}} x f_V(x) Pr(W < 1/x) \delta x$$

b_0 and b_r are fixed values (maximum and minimum values of V). So to minimize the cost, the negative part should be maximized. While this is easy for some distributions like the uniform distribution, it is not straight forward for others. In our experiments we estimate the distributions of interest by histograms and solve the above optimization problem numerically.

In deriving the previous optimization equations, we assumed that only one list is under process. However, in a real scenario often several lists are being processed and the stopping condition depends on all of them. Therefore $g(f_{w_j}(b_{ji})) < 1$ is not a good estimate of the stopping condition. We treat the lists independently and use the maximum number of terms a profile can have (m') to estimate the stopping condition by $g(f_{w_j}(b_{ji})) < 1/m'$ instead. To account for this change in the above equations, $Pr(W < 1/x)$ should be replaced with $Pr(W < 1/xm')$.

5.4 Result Maintenance

So far we have considered the problem of efficient profile filtering when a new document arrives. Another important aspect of our streaming scenario is the sliding window which specifies the valid documents. In this section we first describe the challenges caused by this temporal factor and then describe our solution.

When a document which is part of the top-k results of a profile expires as the window slides, another document from the existing valid documents should replace it. The process of re-evaluating a top-k query usually incurs high cost on the system. Given the fact that we aim at supporting a large number of profiles, this cost can slow down the system and prevent it from timely and correct

responses to the users. This problem is closely related to view maintenance discussed in the database community [YYY+03].

In the following, we consider a given profile p and when we mention the score of a document or the top-k results, it is with regard to this specific profile. Let us assume a set $p.R$ of documents is maintained for p. To avoid ever re-evaluating p over the set of *all* valid documents, $p.R$ should contain all documents which have a chance of becoming a top-k result in their life time. This set consists of the top-k current results as well as documents which have a smaller score or shorter life time compared to at most $k-1$ other documents. This concept has been previously exploited in the context of continuous top-k processing in [MBP06] and in kNN queries in [BOPY07]. For a given profile p, we consider the documents in the time/score space where score corresponds to the similarity degree of a document and p, and *time* presents its arrival time. A document d_1 dominates d_2 if $d_1.time > d_2.time$ and $d_1.score > d_2.score$. The k-skyband [MBP06] of a set of points is a subset of these points where each is dominated by at most $k-1$ other points. Clearly if a document is not in the k-skyband it can never be a top-k result of p, as at least k documents with higher similarity grades and longer life times exist.

While many new documents do not qualify as relevant results to a profile due to their low similarity degrees, they are part of the k-skyband as a result of their *time* dimension: since we are considering *in order* streams, *all* incoming documents are part of the k-skyband of *all* profiles at the time they arrive. Such documents remain in a profile's k-skyband only for a short amount of time until they are dominated by fresher, more relevant documents. Inserting each incoming document to all profiles' k-skybands, incurs a large space overhead as well as unnecessary CPU cost to actually maintain the k-skyband.

To circumvent these costs, we restrict the documents which are inserted to $p.R$ to those which have scores larger than a threshold τ and maintain the k-skyband over them. We call this part of the k-skyband the ***horizon***. With suitable values of τ, the horizon is expected to be more stable, i.e. its members do not disqualify frequently, and more promising, i.e. its members are more likely to actually become a top-k result. In the following we first describe our *horizon result maintenance* method and then discuss suitable values of τ.

Consider a profile p and its corresponding set of documents $p.R$. A re-evaluation over the set of all valid documents is invoked when $|p.R| < k$ and in this case $p.R$ is set equal to the obtained top-k results. A newly arrived document d is inserted to $p.R$ if $sim(d,p) \geq \tau$. When a new document is inserted in $p.R$ the dominance values (i.e., a counter) of existing documents are updated accordingly and those documents whose dominance value hits k are eliminated from $p.R$. Note that removing a document from $p.R$, either due to expiration or as a result of dominance by k other documents, does not affect the dominance values of other existing documents: all documents dominated by this document should have been removed before.

Fixing τ to a predefined static value is not suitable for our dynamic setting as an appropriate value currently may be too small or big in future with regard to

```
Input: newdocs,expireddocs
foreach document d ∈ expireddocs do
    D.remove(d);
    foreach profile p ∈ d.profiles do
        p.R.remove(d);
        if |p.R| < k then
            re-evaluate(p);
            p.updateTopK(p.R.topK);
        end
    end
end
tobeUpdated = ∅;
foreach document d ∈ newdocs do
    D.insert(d);
    tobeUpdated = ProfileFilter(d);
    foreach profile p ∈ tobeUpdated do
        p.R.insert(d);
        p.R.updateDominance();
        if p.R.topK has changed then
            p.updateTopK(p.R.topK);
        end
    end
end
```

Algorithm 4: The overall Algorithm for removing expired documents and inserting new documents

the corresponding window of valid documents. A too small value would result in all documents qualifying for insertion to $p.R$ and ultimately maintaining the complete k-skyband. On the other hand, a too large value causes $p.R$ to frequently contain less than k documents and firing numerous re-evaluations. A dynamic value for τ which adapts to the relevance of current documents is the remedy.

Let $p.R.score$ denote the score of the ranked last document in $p.R$. We show that for any value of τ smaller than or equal to $p.R.score$, $p.R$ contains the correct top-k results: the top-k results in $p.R$ are the same as the result of evaluating p over all valid documents. We also show, by a contradicting example that this is the largest value which still guarantees the correctness of the horizon. Note that the *correctness* concern raises due to dynamically changing τ.

Theorem 4 *Let $p.R.score$ denote the similarity score of the ranked last document in $p.R$. If τ is changing dynamically, the necessary and sufficient condition for $p.R$ to contain the correct top-k results is that $\tau \leq p.R.score$.*

For the proof please see the appendix.

5.4.1 Integration with Profile Filtering

To integrate the above described horizon maintenance scheme with the proposed profile filtering algorithm, the $p.s$ values used in calculating $p.v$ in Section 5.3 should be replaced with τ. This will ensure that the profile filtering algorithm will not miss any profiles p where a new document should be inserted to $p.R$, although the document may not qualify as a top-k result of p currently. The overall steps for inserting documents and updating profiles are shown in Algorithm 4. D denotes the set of valid documents. First, expired documents are removed from D. $d.profiles$ denotes the affected profiles by d: all profiles p where $d \in p.R$. An affected profile p of an expired document is re-evaluated if $|p.R| < k$. Then, for each of the incoming new documents the profile filter returns the profiles which should be updated with this document. Note that if document d is inserted in $p.R$ it is not necessarily a top-k result of p, but it is part of p's horizon.

5.5 Experiments

We have implemented a simulation of the envisioned system in Java 1.6. The dataset is stored in an Oracle 11g database and replayed according to the timestamps attached to the entries.

Dataset and Profiles

We have obtained the ICWSM 2009 Spinn3r Blog Dataset[3]. It consists of 44 million blog posts between the time period of August 1st and October 1st, 2008. Each blog entry (post) consists of plain text, a timestamp, a set of tags, and other meta information such as the blog's homepage URL. The data is formatted in XML and is further arranged into tiers approximating to some degree search engine ranking. We have parsed the blog posts for the highest tier levels resulting in $2,444,780$ distinct posts. After stemming and stopword removal, we have inserted the content of each blog as $\langle term, score \rangle$ pairs in the database where the TF/IDF methodology is used for assigning weights.

Profiles are generated by looking at frequently used topic descriptions of the blog entries, such as "US election". Each profile has between 3 and 5 out of 657 distinct terms and their corresponding weights are chosen uniformly at random. We did not use one of the standard search engine query logs as subscription queries are of a more general nature. We use the well-accepted cosine measure to calculate the similarity degree between a document and profile. Note that as mentioned in Section 5.1.1 the cosine measure has the two necessary properties of monotonicity and homogeneity. We assume that the document and profile vectors are normalized by their lengths: $|p| = |d| = 1$, so $sim(d, p) = \sum_{i=1}^{m} w_i u_i$.

[3] http://www.icwsm.org/2009/data/

Parameter	Default	Range
number of profiles	50K	30,40,50,60,70,80
result cardinality (k)	10	5,10,15,20
window size	4500(s)	2700,3600,4500,5400,6300
number of groups	10	2,4,6,8,10,12,14

Table 5.1: Variations of the parameters as used in the experiments.

Algorithms and Measures of Interest

We consider the following three algorithms for profile filtering.

- **naive:** As a baseline we have implemented a profile filtering algorithm that keeps non-sorted inverted lists of profiles and reads, for an incoming document, all entries from all inverted lists that correspond to a term in the document.

- **col:** This is our algorithm as described in Section 5.3 which keeps all profiles in term-based index lists, sorted by score. The exact ordering is kept at all times, which has benefits for the profile filtering process but comes with the cost of placing or re-placing entries to the exact position w.r.t. their scores.

- **pol:** This algorithm as described in Section 5.3.1 divides the inverted lists to groups, maintaining the group membership criteria for the entries but not the order among entries of a particular group. We expect a larger percentage of profiles read during the profile filtering but a significantly lower maintenance cost.

For all the above we have two alternatives for result maintenance: (i) using a simple top-k list and re-evaluating whenever one of the top-k results expires or (ii) maintaining each profile's *horizon*, as explained in Section 5.4.
We have also implemented **the approach by Mouratidis et al.** in [MP09], described in Section 3.2.2, which we will refer to it as **incr.-thresh**.

We report on CPU time as our main measure of performance. Note that we do not report on accuracy measures as all algorithms report the exact top-k results. To better understand the effects of our proposed algorithms we have measured CPU time for different parts of the algorithms, in addition to the overall time. Also for the result maintenance algorithm we are interested in the space overhead imposed by retaining more than k documents per profile.

Depending on the sliding window size, the number of documents inserted in the system, ordered on their timestamps, is such that at least 5 non-overlapping sliding windows have completed. All measurements are averaged over all processed documents after a warm-up phase of 500 documents. The parameters, their default values and range of variations are shown in Table 5.1. All algorithms are run on a quad-core Intel Xeon CPU E5530 @2.4 GHz, 48 GB main memory, 4 TB of hard drive and Microsoft Windows Server 2008 R2 x64 as operating system.

Algorithms	total	profile fil.	update	re-eval	insert
naive-k	23.12	7.45	0.00	14.73	0.67
col-k	23.87	2.36	3.12	17.39	0.74
pol-k	21.14	3.48	0.52	16.17	0.71
naive-horizon	11.01	7.18	0.00	1.71	1.80
col-horizon	9.64	3.47	1.30	2.71	1.83
pol-horizon	8.84	4.60	0.28	1.78	1.85
incr.-thresh	13.86	-	-	-	-

Table 5.2: Average time measurements (ms).

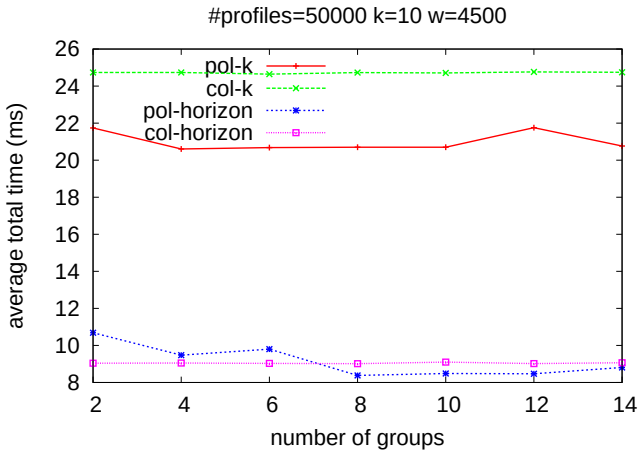


Figure 5.2: The effect of number of groups on average total time.

Results

Table 5.2 shows the time measurements for the default system parameters for naive, col, pol and, incr.-thresh. Total time includes the time spent for removing old documents, updating the affected profiles by re-evaluating them, profile filtering for a new incoming algorithm and inserting it to the result sets of selected profiles. To have a better understanding of the effect of different approaches, Table 5.2 shows the time spent for each of these parts separately. Note that we do not show the time which is the same for all algorithms, like the time to insert a document in the term-document inverted list, but this is included in the total time. Furthermore, for incr.-thresh we only show the total time, as this algorithm does not have same separated modules for the above mentioned tasks. The first observation is that a significant portion of time (31%) is spent in the profile filtering component in the naive-k algorithm. The col-k algorithm decreases this time by almost 68% at the expense of large update time for keeping its necessary structures up-to-date. Our proposed pol-k algorithm is successful in decreasing the time spent for profile filtering as well as limiting the update cost. Note that col-k has a bigger re-evaluation time, as re-evaluating a profile

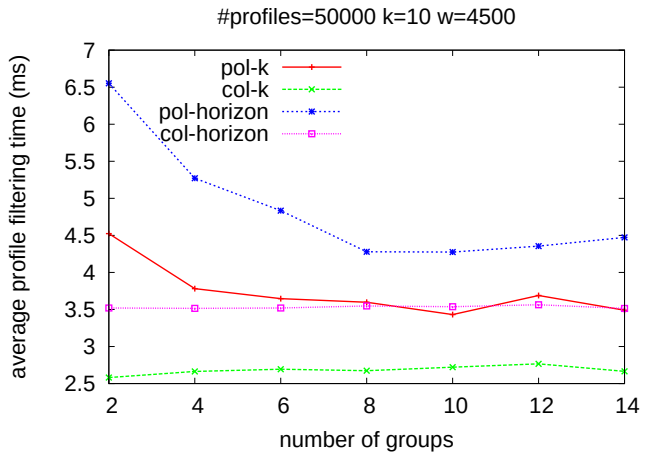


Figure 5.3: The effect of number of groups on average profile-filtering time.

causes its $p.s$ value to change, causing updates in the inverted lists which should be kept sorted for col-k. While col-k incurs a larger total time due to its huge update cost, pol-k achieves in total 8% improvement compared to naive-k. The next three rows of this table show the measurements for the horizon variation of the algorithms. The re-evaluation cost decreases for all algorithms by almost 84% at the expense of a relatively small increase in the result insertion time. In the horizon variations, result insertion is more costly as it involves updating the dominance counters and k-skyband maintenance. Since with the horizon method more profiles get qualified to have a document in their result set, we observe an increase in the profile filtering time for col-horizon and pol-horizon compared to their top-k counterparts. However, since the ranked last document in the result set, which defines the values of interest for keeping the lists ordered, changes less frequently than in the top-k method, the update cost for these algorithm decreases significantly. In total, we observe 60% decrease in the total time, from naive-k to pol-horizon which achieves the smallest total time among all algorithms. incr.-thresh inserts all documents that are in any index list above the scan line of that profile which causes the result set to grow very large. This has the benefit of eliminating the re-evaluations, but on the downside large space is consumed and a large result set should be kept sorted which incurs extra cost. Overall, pol-horizon has a decrease in total time of 36% together with significant decrease in the resultset size it maintains compared to incr.-thresh.

As seen previously, the pol algorithm is successful in maintaining the decrease in profile filtering time as well as limiting the time spent for updating the required structures. The calculations necessary for choosing the boundary

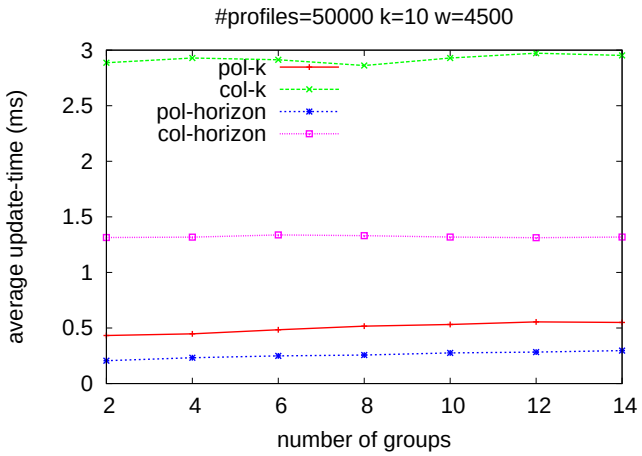


Figure 5.4: The effect of number of groups on average update time.

values mentioned in Section 5.3.2 are performed only once after the warm-up phase. Figures 5.2,5.3, and 5.4 present the effect of number of groups. Figure 5.2 shows the total time for pol and col with the top-k and horizon variations. We observe that with as small as 10 groups, pol achieves very good decrease in the total time. In Figures 5.3 and 5.4 we observer the profile filtering and update times for different number of groups. pol-k incurs almost 6 times less update cost compared to col-k, at the expense of small increase in its profile filtering time. As mentioned in the previous paragraph, the horizon variations have smaller update cost and slightly larger profile filtering time.

The effect of number of profiles on total time is shown in Figure 5.5. The total time for all algorithms increases with increasing the number of profiles, as the profile filtering and re-evaluation parts become more costly. However, the effect of our profile filtering algorithms are more visible for larger number of profiles. Also note that pol does not show any significant drop in decreasing the total time compared to col, although the number of groups are fixed for all profile cardinalities to 10. This is because by using the horizon maintenance module, the update cost in col decreases significantly, as shown in Figure 5.4. incr.-thresh has much larger total time, since similar to col-k it spends a lot of time updating the index lists' scan lines. The pol-horizon algorithm achieves the minimum total time, decreasing it by up to 40% from incr.-thresh

We report on the effect of sliding window size on average total time in Figure 5.6. The average total time decreases with increasing size of the sliding window for all algorithms. This is mainly due to the decrease in number of necessary re-evaluations on average. With a bigger window size, high quality documents live longer and a larger time span allows for high quality documents to arrive

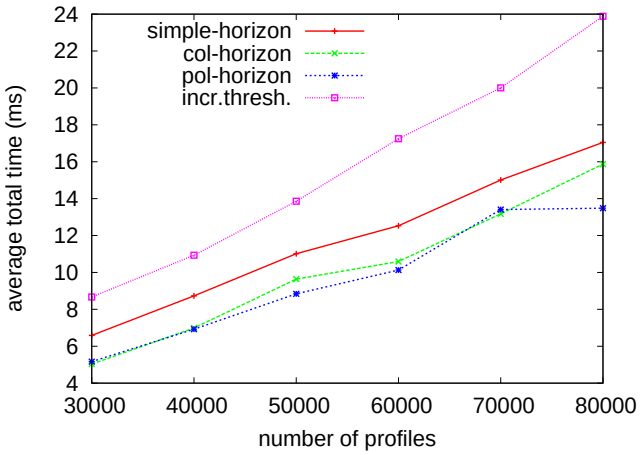


Figure 5.5: The effect of the number of profiles on average total time.

k	#reeval top-k	size top-k	#reevals horizon	size horizon
5	73.18	4.99	11.49	7.93
10	141.03	9.99	6.56	17.84
15	209.00	14.97	3.55	28.75
20	278.51	19.94	2.62	40.11

Table 5.3: Number of re-evaluations and result set sizes when changing k. w=4500(ms) and #profiles = 20000.

before others expire and fire a re-evaluation. As seen in the Figure, with large enough window sizes pol-horizon and col-horizon have similar total time which is the result of fewer updates.

Table 5.3 reports on the average number of re-evaluations and result set size for the top-k and horizon variations. Note that the profile filtering algorithm does not have an effect on these values so we have not repeated the results by separately reporting on them. First, note that the necessary number of re-evaluations drops from 7 to almost 100 times less for the horizon method compared to top-k based maintenance. The very interesting observation is that with increasing the k value, the number of re-evaluations has an increasing trend for the top-k method but a decreasing one for the horizon algorithm. This is because for larger values of k the horizon grows much larger than k, significantly decreasing the chance of the result set containing less than k results to fire a re-evaluation. However, the horizon method comes with the extra cost of maintaining the horizon.

In summary we observe that the pol-horizon combination offers significant performance gains compared to the rest of algorithms. The horizon result maintenance algorithm causes small decrease in the improvement pol can offer in decreasing the profile filtering time. However, it drastically decreases the necessary update cost and number of re-evaluations while incurring only a small

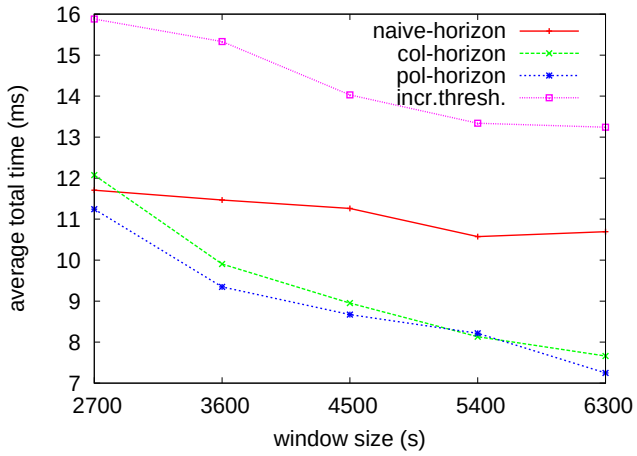


Figure 5.6: The effect of sliding window size on average total time.

space over head over the system. A decrease of between 25% to 30% in overall processing time, allows our envisioned system to scale better to larger number of profiles and higher data rates.

5.6 Conclusion

Motivated by the tremendous popularity of blogs, micro-blogging services like Twitter, or online newspapers, we address the problem of continuously processing a massive amount of user defined subscription queries (profiles) over a stream of documents. The challenge in processing these queries in real-time lies not only in the fact that there are many queries, but also, and foremost, in the observation that data streams in at high rates. Both properties combined call for a careful profile filtering process, that omits evaluating too many profiles. Our approach reduces the number of necessary profiles which have to be evaluated at data arrival time, by organizing the user profiles in a so called inverted index, where, for each term we store a sorted list of profiles that contain this term. The key idea is to sort profiles not only on their weight w.r.t. a term but also according to the quality of the currently alive documents which are ranked high for the particular profile. This sorting criteria allows for an effective stopping condition for the profile filtering algorithm. Furthermore, we observe that keeping the entire lists completely sorted is infeasible, due to the high data arrival rates, profiles move up and down in the lists as time evolves, requiring permanent updates to the lists. We address this by using group sorted lists, i.e., lists consisting of different groups which are sorted relatively to each other, but without order inside groups. As the definition of the group boundaries is crucial for the overall performance gain, we present a method to select these

bounds by leveraging score distribution information derived from histograms. We combine our proposed filtering algorithm with an effective skyline-based result maintenance algorithm. Since we are considering Web data, the stream of incoming data can be considered to be in-order. This property of the incoming stream causes many low scored documents to be unnecessarily considered as part of the result set, only due to their freshness. Our *horizon* result maintenance module constraints the documents which can enter the result set without trading the accuracy of results. The horizon algorithm does not depend on the order of incoming data and similar to the skyline approach, can tolerate any degree of being out-of-order. Combining our proposed filtering algorithm the horizon result maintenance algorithm, which cuts drastically on the number of necessary re-evaluations caused by expiring documents, we observe a further decrease in the update costs of the access structures. We evaluate our approach using a real world blog dataset demonstrating the performance gains compared the state-of-the-art.

Chapter 6

Distributed KNN Search Over High Dimensional Data

6.1 Introduction

The rapid growth of online information, triggered by the popularity of the Internet and the huge amounts of user-generated content from Web 2.0 applications, calls for efficient management of this data to improve usability and enable efficient and accurate access to the data. User-generated data today range from simple text snippets to (semi-) structured documents and multimedia content. To enable rich representation and avoid loss of information, the number of features extracted to represent the data is very often high. Furthermore, as the data sources are naturally distributed in large-scale networks, traditional centralized indexing techniques become impractical. To address the demanding needs caused by this rapidly growing, large-scale, and naturally distributed information ecology, we propose in the following an efficient, distributed, and scalable index for high-dimensional data enabling efficient and accurate similarity search.

Peer-to-Peer (P2P) overlay networks are well-known to facilitate the sharing of large amounts of data in a decentralized and self-organizing way. These networks offer enormous benefits for distributed applications in terms of efficiency, scalability, and resilience to node failures. Distributed Hash Tables (DHTs) [SMK+01, RFH+01] (c.f., Chapter 2.1.2), for example, allow efficient key lookups in logarithmic number of routing hops but are typically limited to exact or range queries. Similarity search in high dimensional data has been a popular research topic in the last years [BGRS99, GIM99, YOTJ01, BBK98, DIIM04]. Distributed processing of such queries is even more complicated, but is unavoidable due to the inherently distributed way data is generated in the Web. Existing approaches to the similarity search problem in high dimensional

data either focus on centralized settings, as cited above, rely on preprocessing data centrally, assume data ownership by peers in a hierarchical P2P setting or fail at providing both high quality search results and a fair load balance in the network [FGZ05, SEAA04, DVKV07].

In this Chapter we consider similarity search over high dimensional data in structured overlay networks. Inspired by the idea of Locality Sensitive Hashing (LSH) technique [GIM99, DIIM04] which probabilistically assigns similar data to the same bucket in a hash-table, we first investigate the difficulties of directly applying this method to a distributed environment and then devise two locality preserving mappings which satisfy the identified requirements of bucket placement on peers of a P2P network. The first requirement, placing buckets which are likely to hold similar data on the same peer or its neighboring peers, aims at minimizing the number of network hops necessary to retrieve the search results, causing a decrease in both network traffic and the overall response time. The second requirement considers load balancing and is satisfied by harnessing estimates of the distribution of resulting data (bucket) mapping.

6.1.1 Problem Statement and System Overview

In similarity search objects are characterized by a collection of relevant features and are represented as points in a high dimensional space. In some applications the objects are considered in a metric space where only a distance function is defined among them and the features of the objects are unknown. However, with the advances in metric space embedding (*cf.*[ABN06]) a vector space assumption is valid and realistic. Given a collection of such points and a distance function between them, similarity search can be performed in the following two forms:

- *K-Nearest Neighbor (KNN) query:* Given a query point q the goal is to find the K closest (in terms of the distance function) points to it.
- *range query:* Given a query point q and a range r the goal is to find all points within a distance r of q.

In many applications returning the approximate KNN of a point, instead of the exact ones, suffices. The approximate version is even more desirable when the data dimensionality is high, as similarity search is very expensive in such domains. Here, the goal is to find K objects whose distances are within a small factor $(1+\epsilon)$ of the true K nearest neighbors' distances. The quality of similarity search is measured by the number of returned results, as well as the distances to the query for the K points returned compared to the corresponding distances of the true K nearest objects for KNN queries.

We consider similarity search in structured peer-to-peer networks, where N peers $P_1, ..., P_N$ are connected by a DHT that is organized in a cyclic ID space, such as in Chord [SMK+01]. Every node is responsible for all keys with identifiers between the ID of its predecessor node and its own ID. Our underlying similarity search method is probabilistic and relies on building several indices of data to achieve highly accurate query results. We assume each of these data

indices is maintained by a subset of size n of all peers where n is an order of magnitude smaller than N. Each of these subsets form a *local* DHT among themselves. For each replica of the data we deterministically select a subset of peers to hold the corresponding index, i.e., not all peers hold a share of the index from the beginning. These initially selected peers are gateway peers to the corresponding local DHTs. The number of peers/nodes inside each local DHT might grow/shrink over time depending on the load of the system. The locations (IDs) of the gateway peers is global knowledge based on a deterministic sampling process with fixed seed value. However, not the peers are known but only their IDs in the underlying network. We explain this process in more detail in Section 6.3. If a gateway peer is not accessible, the peer currently holding the gateway peer ID is asked to join the local DHT using one of the other gateway peers. These dynamics are handled by the underlying network which we do not address in this thesis.

Our goal is to map the high dimensional data to the peers in a way that assures fair load balancing in the local DHTs and at the same time enables efficient and accurate KNN and range query processing.

This Chapter is based on work presented in [HMCMA08, HMA09a] and is organized as follows. Section 6.2.1 presents the requirements of distributing LSH indices to the linear peer domain and puts forward two mappings which satisfy those requirements. Section 6.3 concentrates on the creation of the local DHTs. Section 6.4 presents the KNN query processing algorithms. Section 6.5 addresses the challenges in processing range queries and presents an approach based on range sampling to overcome these constraints. Section 6.6 presents the experimental evaluation of our approach. Section 6.7 concludes this Chapter.

6.2 Mapping LSH to the peer identifier space

6.2.1 Revisiting LSH

The basic idea behind the *LSH-based* approaches is the application of *locality sensitive hashing* functions. A family of hash functions $H = \{h : S \rightarrow U\}$ is called (r_1, r_2, p_1, p_2)-sensitive if the following conditions are satisfied for any two points $\mathbf{q}, \mathbf{v} \in S$:

- if $dist(\mathbf{q}, \mathbf{v}) \leq r_1$ then $Pr_H(h(\mathbf{q}) = h(\mathbf{v})) \geq p_1$
- if $dist(\mathbf{q}, \mathbf{v}) > r_2$ then $Pr_H(h(\mathbf{q}) = h(\mathbf{v})) \leq p_2$

where S specifies the domain of points and $dist$ is the distance measure defined in this domain.

If $r_1 < r_2$ and $p_1 > p_2$, the salient property of these functions results in more similar objects being mapped to the same hash value than distant ones. The actual indexing is done using LSH functions and by building several hash-tables to increase the probability of collision (*i.e.* being mapped to the same hash value) for close points. At query time, the KNN search is performed by hashing

the query point to one bucket per hash-table, scanning that bucket and then ranking all discovered objects by their distance to the query point. The closest K points are returned as the final result.

In the last few years, the development of locality sensitive hash functions has been well addressed in the literature. In this work, we consider the family of LSH functions based on p-stable distributions [DIIM04]. This family of LSH functions are most suitable when the distance measure between the points is the l_p norm. Given a point $\mathbf{v} = v_1, \ldots, v_2$ in the d-dimensional vector space, its l_p norm is defined as: $||\mathbf{v}||_p = (|v_1|^p + ... + |v_d|^p)^{1/p}$.

Stable Distribution: A distribution D over $\mathbb{R}$ is called p-stable, if there exists $p \geq 0$ such that for any n real numbers $r_1 \ldots r_n$ and i.i.d. variables $X_1 \ldots X_n$ with distribution D, the random variable $\sum_i r_i X_i$ has the same distribution as the variable $(\sum_i |r_i|^p)^{1/p} X$, where X is a random variable with distribution D. p-stable distributions exist for $p \in (0, 2]$. The Cauchy and Normal distributions are respectively 1-stable and 2-stable.

In the case of p-stable LSH, for each d-dimensional data point $\mathbf{v}$ the hashing scheme considers k independent hash functions of the following form :

$$h_{\mathbf{a},B}(\mathbf{v}) = \lfloor \frac{\mathbf{a} \cdot \mathbf{v} + B}{W} \rfloor \tag{6.1}$$

where $\mathbf{a}$ is a d-dimensional vector whose elements are chosen independently from a p-stable distribution, $W \in \mathbb{R}$, and B is drawn uniformly from $[0, W]$. Each hash function maps a d-dimensional data point to an integer. With k such hash functions, the final result is an integer vector of dimension k of the the following form:

$$g(\mathbf{v}) = (h_{\mathbf{a_1},B_1}(\mathbf{v}), ..., h_{\mathbf{a_k},B_k}(\mathbf{v})) \tag{6.2}$$

In this work we assume the distance function is the widely used l_2 norm (Euclidean distance) and use the Normal distribution as our p-stable distribution.

In LSH-based schemes, in order to achieve high search accuracy, multiple hash-tables need to be constructed. Experimental results [GIM99] show that the number of hash-tables needed can reach up to over a hundred. In centralized settings this causes space efficiency issues. While this constraint is less visible in a P2P setting, a high number of hash-tables results in another serious issue arising specifically in this environment. In order to visit all hash-tables (which is needed to answer the KNN query with good accuracy) a large number of peers may need to be contacted. Solutions to this shortcoming in centralized settings [LJW+07, Pan06] suggest investigating more than one bucket in each hash-table, instead of building many different hash-tables. The main idea is that we can guess which buckets other than the bucket which the query hashes to, are more likely to hold data that is similar to the query point. In our envisioned P2P scenario, jumping from one bucket to another can potentially cause jumping from one peer to another, which induces $O(log\, n)$ network hops in a network of n peers. In the following Section, we discuss and introduce mapping schemes which allow us to significantly reduce the number of incurred network hops during query time by grouping those buckets which are likely to

hold similar data on the same peer, while effectively balancing the load in the network.

Given the output of the p-stable LSH, which is a vector of integers, we consider a mapping to the peer identifier space, denoted as $\xi : \mathbb{Z}^k \rightarrow \mathbb{N}$.

Different instances of the mapping function ξ come with different characteristics w.r.t. to load balancing and the ability to efficiently search the index. In terms of network bandwidth consumption and number of network hops, clearly, a mapping of all data to just one peer is optimal. Obviously, this mapping suffers from a huge load imbalance. The other extreme is to assign each hash bucket to a peer using a pseudo-uniform hash function that provides perfect load balancing but steals any control on grouping similar buckets on the same peer, therefore causing an excessive number of DHT lookups. More formally, ξ should satisfy the following two conditions:

- *Condition 1:* assign buckets likely to hold similar data to the same peer.
- *Condition 2:* have a predictable output distribution which fosters fair load balancing.

Figure 6.1 shows an illustration of the overall mapping from the d-dimensional space, to the k-dimensional LSH buckets, to finally the peer identifier space using ξ.

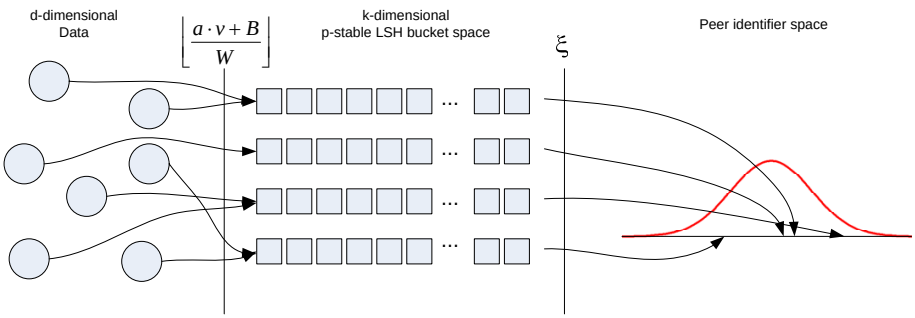


Figure 6.1: Illustration of the two level mapping from the d-dimensional space to the peer identifier space.

We first try to capture the semantics of similar buckets: *buckets likely to hold close data*. The first condition of the LSH definition states that close points are more likely to be mapped to the *same* hash value. However, it is not clear which hash buckets are more probable to hold similar data. This has been discussed also in [LJW+07, Pan06] in a query-dependent way. However we need a more general view, as mapping buckets to peers should be independent of queries. We show that using hash functions of the form of Equation 6.1 close points have a higher probability of being mapped to *close* integers, that is, integers with small l_1 distance. This is more general than the LSH definition, i.e. being hashed to the *same* value. Since bucket labels are concatenations of such integers, we argue that the l_1 distance can capture the distance between

buckets, *buckets likely to hold close data have small l_1 distance to each other.* We prove the following theorems following the above argument.

Theorem 5 *For any three points $\mathbf{v_1}, \mathbf{v_2}, \mathbf{q} \in S$ where $||\mathbf{q} - \mathbf{v_1}||_2 = c_1$ and $||\mathbf{q} - \mathbf{v_2}||_2 = c_2$ and $c_1 < c_2$ the following inequality holds:*

$$pr(|h(\mathbf{q}) - h(\mathbf{v_1})| \leq \delta) \geq pr(|h(\mathbf{q}) - h(\mathbf{v_2})| \leq \delta)$$

For the proof see the appendix.

Theorem 6 *For any two points $\mathbf{q}, \mathbf{v} \in S$, $pr(|h(\mathbf{q}) - h(\mathbf{v})| = \delta)$ is monotonically decreasing in terms of δ.*

For the proof see the appendix. The above two theorems indicates that l_1 can capture the distance between buckets in terms of probability of holding close data: Given bucket labels $\mathbf{b_1}$,$\mathbf{b_2}$ and $\mathbf{b_3}$ which are integer vectors of dimension k, if $||\mathbf{b_1} - \mathbf{b_2})||_1 < ||\mathbf{b_1} - \mathbf{b_3}||_1$, then $\mathbf{b_1}$ and $\mathbf{b_2}$ have a higher probability to hold similar data than $\mathbf{b_1}$ and $\mathbf{b_3}$.

Having a better understanding of the semantics of similar buckets, we now discuss two mappings which satisfy the two conditions mentioned above.

6.2.2 Linear Mapping based on Sum

We propose $\xi_{sum}(\mathbf{b}) = \sum_{i=1}^{k} b_i$ as an appropriate placement function which can be used to map the k-dimensional vector of integer $\mathbf{b}$, as the output of p-stable LSH, to the one dimensional peer identifier space. The intuition is that the sum treats all bucket label parts b_i equally and that minor differences in the b_i values are smoothed out by the sum leading to close $\xi_{sum}()$ values for "close" bucket labels. In the following, we show how relying on p-stable LSH and its characteristics, it satisfies both conditions above.

We first investigate condition 1. As discussed in the previous Section, buckets which are likely to hold similar data have small l_1 distance to each other. Given ξ_{sum} as our mapping function we have: $|\xi_{sum}(\mathbf{b_1}) - \xi_{sum}(\mathbf{b_2})|=|(b_{11} - b_{21}) + \cdots + (b_{1k} - b_{2k})| \leq ||\mathbf{b_1} - \mathbf{b_2}||_1$. Which means if buckets with labels $\mathbf{b_1}$ and $\mathbf{b_2}$ are likely to hold similar data, $\xi_{sum}(\mathbf{b_1})$ and $\xi_{sum}(\mathbf{b_2})$ will also be close. Given the assignment of data to peers in Chord-style overlays, this results in assigning the two buckets to the same or neighboring peers in the Chord ring with high probability.

As for the second condition, assume d-dimensional points, $\mathbf{a}$ and $\mathbf{v_1}$. If elements of $\mathbf{a}$ are chosen from a Normal distribution with mean 0 and standard deviation 1, denoted as *N(0,1)*, $\mathbf{a} \cdot \mathbf{v_1}$ is distributed according to the Normal distribution $N(0, ||\mathbf{v_1}||_2)$. For not too large W, $h_{\mathbf{a},B}(\mathbf{v}_1)$ is distributed according to the Normal distribution $N(\frac{W}{2W}, \frac{||\mathbf{v_1}||_2}{W})$ where h is function of the form Eq. 6.1. Therefore $g(\mathbf{v_1})$ will be a k-dimensional vector, whose elements follow

the above distribution. We can now benefit from a nice property of the Normal distributions under summation: $\xi_{sum}(\mathrm{g}(\mathbf{v_1}))$ is distributed according to the Normal distribution

$$N(\frac{k}{2}, \frac{\sqrt{k}||\mathbf{v_1}||_2}{W})$$

The global picture consisting of all data points $\mathbf{v_1}, \ldots, \mathbf{v_M}$ first projected using p-stable LSH and then mapped to $\mathbb{Z}$ by ξ_{sum}, following the Normal distribution

$$N(\frac{k}{2}, \frac{\sqrt{k\sum_i ||\mathbf{v_i}||_2^2}}{\sqrt{M}W})$$

We can therefore predict the distribution of the output of ξ_{sum}, having an estimate of the mean of data points' l_2 norm. We assume that we know the mean norm of available data, but as we will later see, this assumption is only relevant for the start-up of the system where gateway peers are inserted into the hash-tables. Calculating statistics, like in our case the mean, over data distributed in large-scale systems has been well addressed in the literature (cf., e.g., [JMB05]). In Section 6.3 we show how this can be used to balance the load in the network.

6.2.3 Linear Mapping based on Cauchy LSH

As another instance of ξ we propose the LSH function based on Cauchy distribution (1-stable) which offers a probabilistic placement of similar buckets on the same peer. More formally, for a bucket label $\mathbf{b}$,

$$\xi_{lsh_{\mathbf{a}',B'}}(\mathbf{b}) = \lfloor \frac{\mathbf{a}' \cdot \mathbf{b} + B'}{W_2} \rfloor$$

where elements of $\mathbf{a}'$ are chosen independently from a standard Cauchy distribution with probability distribution function

$$cr(x; x_0, \gamma) = \frac{1}{\pi\gamma} \frac{1}{1 + (\frac{x - x_0}{\gamma})^2}$$

where the *location* parameter $x_0 = 0$, *scale* parameter $\gamma = 1$, $W_2 \in \mathbb{R}$, and B' is chosen uniformly from $[0, W_2]$. Note that $\mathbf{b}$ denotes a bucket label and is a k dimensional vector of integers which is itself the output of an LSH function applied on a d-dimensional data point. Given that this LSH function is most suitable for the l_1 norm and that l_1 captures the dissimilarity among buckets, $\xi_{lsh_{\mathbf{a}',B'}}$ probabilistically satisfies condition 1.

The output distribution of this function is similarly predictable. Assume a bucket label $\mathbf{b_1}$. Given the characteristics of p-stable distributions, $\mathbf{a}' \cdot \mathbf{b_1}$ follows the distribution $||\mathbf{b_1}||_1 X$ where X is a Cauchy distribution. For not too large W_2, $h_{\mathbf{a}',B'}(\mathbf{b}_1)$ is distributed with the probability distribution function

$$cr(x; \frac{W_2}{2W_2}, \frac{||\mathbf{b_1}||_1}{W_2})$$

a Cauchy distribution with *location* parameter $\frac{1}{2}$ and *scale* parameter $\frac{||\mathbf{b_1}||_1}{W_2}$. Now, considering all bucket labels, $\mathbf{b_1}, \ldots, \mathbf{b_P}$ mapped to $\mathbb{Z}$ by $\xi_{lsh_{\mathbf{a}',B'}}$, the output follows the Cauchy distribution with *location* parameter $\frac{1}{2}$ and *scale* parameter $\frac{\sum ||b_i||_1}{PW_2}$. In this case, to be able to predict the distribution of the output, we need the mean of l_1 norms of all possible bucket labels. Since the initial hash functions $h_{\mathbf{a},B}$ are known to all peers, this again boils down to the problem of distributed statistics calculation [JMB05].

6.3 Local DHT Creation

To map a particular domain of integer values to a (subset of) peers, it is important to know the size and distribution of the domain. As discussed in Sections 6.2.2 and 6.2.3 the values generated by ξ_{sum} and ξ_{lsh} follow known distributions. Here we describe how this information can be utilized to create local DHTs which as described in Section 6.1.1 maintain the data index.

Consider a linear bucket space of M buckets in which we want to distribute the values generated by the ξ_{sum} mapping. The case for ξ_{lsh} follows similarly. Let μ_{sum}, σ_{sum} be the mean and the standard deviation of the values generated by ξ_{sum} (cf. Section 6.2.1). We choose the first bucket (at position 1) to be responsible for $\mu_{sum} - 2 * \sigma_{sum}$ and the last bucket (at position M) to be responsible for $\mu_{sum} + 2 * \sigma_{sum}$. We restrict ourselves to the span of two standard deviation to avoid overly broad domains and map the remaining data to the considered range via a simple modulo operation:

$$\psi(value) := (\frac{value - (\mu_{sum} - 2 * \sigma_{sum})}{4 * \sigma_{sum}} * M) \, mod \, M \tag{6.3}$$

As mentioned in Section 6.1.1 we want to maintain each particular LSH hash-table (which is an index of the whole data points) by a subset of peers that is usually some orders of magnitude smaller than the global set of peers. To limit the responsibility of maintaining one hash table to a subset of peers, we dynamically form separate local DHTs for each hash-table as follows: At system startup, we place γ peers at predefined positions (known by all peers) based on the normal distribution $N(\mu_{sum}, \sigma_{sum})$ by sampling γ values from $N(\mu_{sum}, \sigma_{sum})$ and mapping them to buckets in the range of $\{1, .., M\}$ using ψ.

For a particular number of initial peers and the sampled values, we consider

$$\rho(value, l) := (\psi(value) + hash(l)) \, mod \, |G|$$

as the mapping of a (value, l)-pair to the global set of peers G, where l is a hash-table id. ρ consists of two components, the previously described ψ function and $hash(l)$ as an offset for global load balancing. The peers responsible for these ρ values are invited to join (create) the particular DHTs.

Algorithm 5 shows the initial algorithm to build up the distributed LSH index. The most important property is the usage of so-called gateway peers (similar to the ones used in [MTW05b]) that are initially placed in each of the

```
Input: Global DHT G, number of gateways γ
init(N(μ_sum, σ_sum));
for (tableId=0; tableId<l;tableId++) do
    sampleSet = ∅;
    for i=0; i<γ; i++ do
        sample = N(μ_sum, σ_sum).nextRandom();
        sampleSet.add(sample);
        P = G.lookup(ρ(sample, tableId));
        if tableId==0 then
            P.createDHT();
        else
            P′ = G.lookup(ρ(sampleSet.getRandom(), tableId));
            P.join(P′);
        end
    end
end
```

Algorithm 5: Initial Algorithm to build up the l hash-tables that contain the gateway peers, drawn from the global peer population

LSH hash-tables. These peers can be determined using the lookup method of the global DHT. If a lookup on one of the predefined positions fails, i.e., leads to a peer that is not currently in the LSH hash-table, that peer issues a lookup on one of the other entry points and joins the particular hash-table it belongs to. In case of a successful access to one of the gateway peers, the query initiator (or data indexing peer) gains access to the LSH hash-table.

The case for ξ_{lsh} is similar, except we use the *location* and *scale* parameters of the predicted Cauchy distribution in Equation 6.3, since mean or standard deviation are not defined for Cauchy distributions. Also at start up, peers are sampled from the predicted Cauchy distribution.

The number of peers dynamically grows inside each local DHT by overloaded peers issuing requests on the global DHT to find peers to join the local DHT on a particular position (bucket). In case of access load problems, the gateway peers can call for a global increment of the number of gateway peers, i.e., increase the number of possible gateway peers that will subsequently be hit by requests and hence invited to join the local DHTs maintaining the LSH hash-tables. We can benefit from the rich related work on load balancing techniques over DHT, such as the work by Pitoura et al [PNT06], that replicates "hot" ranges inside a Chord style DHT and then lets peers randomly choose among the replicated arcs.

6.3.1 Handling Churn

We will now discuss possible ways to handle churn, in particular, peers leaving the system without prior notice, but leave any detailed analysis and in particular the impact of the low level (DHT based) churn handling mechanisms to the

overall performance as future work.

To handle churn it is common practise to introduce a certain degree of replication to the system. One such replication based mechanisms has been already introduced above when presenting the concept of multiple gateway peers per local DHT which solves the following problem: If a peer lookup on one of the predefined positions leads to a peer that is not currently in the local DHT as a gateway peer, that peer issues a lookup on one of the other entry points and joins the particular hash-table it belongs to. We can furthermore add two more ways of replication: (i) replication of complete local DHTs and/or (ii) replication within particular local DHTs. While approach (i) is straight forward to implement it is extremely coarse and more suitable for handling access load problems (hot peers) rather than handling churn in an efficient way. Approach (ii) seems to be more suitable for handling churn: neighboring peers within each local DHT could maintain also the index of their immediate neighbors and in case of a peer failure transmit the replicas to the new peer joining the free spot. Both approaches certainly cause higher load on the system not only in terms of storage but in particular in terms of message exchanges to keep the replicas in sync. We will investigate the impact of replicated local DHTs in our future work and for this paper concentrate on the actual indexing mechanisms. Note that in addition to the replication mechanisms presented above, the underlying global DHT might use replication of routing indices as well, which is treated by us as a black box.

6.4 KNN Query Processing

Given l LSH hash-tables, a query point $\mathbf{q} = (q_1, ..., q_d)$ is first mapped to buckets $g_1(\mathbf{q}) \dots g_l(\mathbf{q})$ using the p-stable LSH method. The query initiator then uses one randomly selected gateway peer per local DHT as an entry to that local DHT. Subsequently, the responsible peer P for maintaining the share of the global index that contains $g_i(\mathbf{q})$ is determined by mapping $g_i(\mathbf{q})$ to the peer identifier space using $\xi(g_i(\mathbf{q}))$, as defined above. The query is passed on to P that executes the KNN query locally using a full scan and passes the query on. We restrict the local query execution to a simple full-scan query processing since we do not want to intermingle local performance with global performance. The local query execution strategy is orthogonal to our work. For the query forwarding (i.e., routing), we consider two possible options: (i) incremental forwarding to neighboring peers or (ii) forwarding based on the multi probe technique [LJW+07]. The results return by all l hash-tables are aggregated at the query initiator and the K closest points to q are returned.

6.4.1 Linear Forwarding

We will now define a stopping condition for the linear forwarding method. Let τ denote the distance of the K^{th} object w.r.t. the query object $\mathbf{q}$, obtained by a local full scan KNN search. Peer P will pass the query and the current

rank-K distance τ to its neighboring peers P_{pred} and P_{succ}, causing each one single network hop. Upon receiving the query, P_{pred} and P_{succ} will issue a local full scan KNN search and compare their best result to τ (cf. Algorithm 6). If the distance d_{best} of the best document is bigger than τ, the peer will not return any results and will stop forwarding the query to its neighbor (depending on the direction, successor or predecessor). The stopping condition can be relaxed by introducing a parameter α and stop forwarding if $d_{best} > \tau/\alpha$. α allows for either a more aggressive querying ($\alpha > 1$) of neighboring peers or for an early stopping ($\alpha < 1$).

```
Input: query q, threshold τ, P_init, direction
result[] = localIndex.executeLocalKnn(q);
if result[0].distance>τ/α then
    done;
else
    resultSet = ∅;
    for (index=0; index<K; index++) do
        if results[index].distance<τ/α then
            resultSet.add(results[index]);
        else
            τ' = resultSet.rankKDistance();
            sendResults(resultSet, P_init);
            forwardQuery(this.predecessor() or/and this.successor(), τ', q,
            P_init, pred or/and succ);
        end
    end
end
```

Algorithm 6: Top-K Style Query Execution based on the locality sensitive mapping to the linear peer space by passing the query on to succeeding or preceding peers.

6.4.2 Multi-Probe Based Forwarding

The multi-probe LSH method [LJW+07] slightly varies the integers in $g(\mathbf{q})$ and produces bucket ID's which are likely to hold close elements to $\mathbf{q}$. For each of these modifications, the method then probes the resulting bucket for new answerers. We adapt this technique as an alternative to the successor/predecessor based forwarding as follows: after the full scan, the peer generates a list of buckets to probe next, considering the maximum number of extra buckets. It is very likely that some of these buckets have already been visited, thus they are removed from the list. For a generated bucket $g(\mathbf{q})$ with $\xi_{sum}(g(\mathbf{q})) \notin]P.pred().id, P.id]$, the peer issues a lookup in the local DHT and forwards the query and bucket list to the peer responsible for $\xi(g(\mathbf{q}))$. The peer that receives the query, issues a full scan, removes visited buckets from the list and forwards the query (cf. Algorithm 7).

```
Input: Local DHT L, query q, bucketlist, P_init
result[] = localIndex.executeLocalKnn(q);
while (bucketlist.hasElement()) do
    b =bucketlist.removeBucket();
    bucketId =ξ(b);
    if bucketId ∈ ]P.pred().id, P.id] then
        nothing to do;
    else
        P_new = L.lookup(bucketId);
        sendResults(P_init, results);
        forwardQuery(P_new, ,bucketlist, P_init);
        break;
    end
end
```

Algorithm 7: Multi Probe based Variant of the KNN query processing.

The multi probe algorithm relies on the parameter that specifies the maximum number of probes, whereas the linear forwarding algorithm has a clear defined stopping condition. The relaxation parameter α is optional.

6.5 Range Query

While LSH provides a nice solution to KNN query processing in high dimensional data, unlike other methods, it is difficult to extend it to range queries. This is because specific LSH functions are designed to map points with certain distance from each other to the same hash value. The parameter r_1 in the definition of LSH functions indicates this distance (cf., Section 6.2.1). Therefore different indices should be made for different parameters r_1 to satisfy range search for different values of the range radius. However it is impractical to construct a different index for each possible range.

With mapping similar buckets to the same peer or to neighboring peers, we can support also range queries. Several buckets, which are likely to hold similar data to the query are investigated.

6.5.1 Range Query Processing

Range query processing over the linearly mapped data is different to the KNN queries as the overall objective is to return *all* items within a particular range, i.e., the stopping condition of the linear forwarding algorithm needs to be adapted. The startup phase of the query processing is the same as described in Section 6.4 for the KNN query algorithms: for a given query we determine the peer responsible for the bucket to which this point is mapped.

Once the starting peer is known and receives the query, it will first execute the query locally using a full scan and return all matching items to the query

initiating peer, i.e., send all items within range $< r$. Subsequently, it will forward the query to its neighboring peers. The forwarding stops at a peer that does not have a single matching item.

While this processing seems to be intuitive it has the following serious drawback caused by an observation illustrated in Figure 6.2. The range of the query is indicated by the long arrow, the spot in the middle represents the query point. Now, the three circles indicate the responsibility of peers for the data items. Obviously, the data in the inner circle can be processed since it is maintained by the initial peer hosting also the query point. Then, however, due to the often naturally clustered data, the following up peer does not maintain any items in the desired range hence causing the algorithm to stop when having covered the range to the second inner circle, thus missing relevant items.

To overcome this problem of "empty" ranges inside the query range, we opt for sampling the range, i.e., starting separate range queries at predefined position.

Unfortunately, the data distribution inside this range is not known a priori. Furthermore, it depends on the query point and on the range itself, which makes it not tractable to pre-compute. This in particular means that even though the actual data distribution inside the hash-tables is known (to the peers maintaining it) it cannot be used to predict the query dependent range of peers to visit.

In absence of any knowledge of data distributions, we employ a simple sampling method that divides the range in equally sized sub-ranges. For each subrange, the query is forwarded to one responsible peer. Subsequently, each peer starts the range query processing and as described above, each processing thread stops when (i) an already queried peer is met or (ii) a peer does not have any single item within the specified range. We will now describe the process of range estimations.

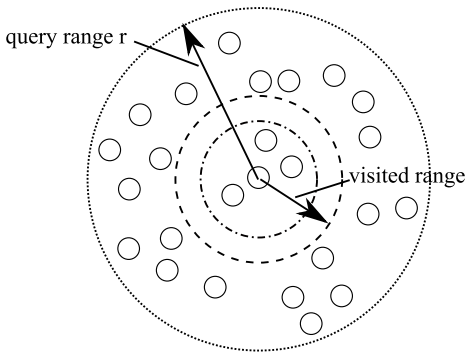


Figure 6.2: Illustration of the problem we face when processing range queries over the linearly mapped data. The "empty" ranges cause the algorithm to stop before the full range has been explored.

6.5.2 Range Prediction

We will now try to derive query dependent lower and upper bounds for the values generated by ξ. As we will later see, these estimates can be used to enable efficient parallelized in-hash-table query processing.

Recall from Section 6.2.1 that the output of the considered LSH hash function is a k-dimensional vector of integer values where each value corresponds to one of the k hash functions of the form of Equation 6.1.

Assume a query point $q = (q_1, ..., q_d)$ and a range r. Let furthermore be $a^{(+)} = argmax_j\{a_{ij}\}$ and $a^{(-)} = argmin_j\{a_{ij}\}$ the positions of the largest and smallest values of elements of one of the k vectors $\mathbf{a}_i$. The idea is to select samples from the d dimensional space that are in distance r of q and produce a bucket label with maximum l_1 difference from $g_{\mathbf{a},b}(\mathbf{q})$ from Equation 6.2. These samples will be mapped into the linear space using $h_{\mathbf{a},b}$ and ξ.

To construct these samples, we repeat the following for all k vectors of $\mathbf{a_i}$. We add to the query vector at the maximal and minimal value positions of vector $\mathbf{a_i}$ the query range r. More formally, we generate the upper range point as $\mathbf{q_i}^{(+)} := \mathbf{q} + \mathbf{j}_{a_i^{(+)}} * r$ and the lower range point as $\mathbf{q_i}^{(-)} := \mathbf{q} - \mathbf{j}_{a_i^{(-)}} * r$ where $\mathbf{j}_i$ is the i^{th} unit vector.

Using these generated samples of points in distance r we apply the standard techniques using LSH hashing and mapping through ξ to determine the upper and lower bound ξ values

$$upper(\mathbf{q}, r) := argmax_i\{\xi(g(\mathbf{q}^{(+)}))\}$$
$$lower(\mathbf{q}, r) := argmin_i\{\xi(g(\mathbf{q}^{(-)}))\}$$

Assume $peer_u$ and $peer_l$ to be the two peers responsible for the above two values. According to condition 1 of an appropriate ξ function, peers which fall between these two peers in the Chord style ring, are most likely to hold data in range r of the query point $\mathbf{q}$. Since the output distribution of ξ is known the above values can be used to estimate the number of peers which should be contacted to answer a query with range r.

6.6 Experiments

6.6.1 Experimental Setup

We have implemented a simulation of the proposed system and algorithms using Java 1.6. The simulation runs on a 2x2.33 GHz Quad-Core Intel Xeon CPU with 8GB RAM. The data is stored in an Oracle 11g database.

Data Sets and Overlay setup

Flickr: We used part of a crawl of Flickr obtained from the Cophir project [BEF+09] consisting of $1,000,000$ images represented by their MPEG7 visual

	Flickr	Corel
#data points	1,000,000	60,000
#dimensions	282	89
#peers in Global DHT (N)	1,000,000	100,000
#peers per Local DHT (n)	1000	100

Table 6.1: Data Sets and Overlay setup

descriptors. The total number of dimensions per image is 282 and contains descriptors such as *Edge Histogram Type* and *Homogeneous Texture Type*. For the global DHT we considered a population of $1,000,000$ peers and each replica of the data set is maintained by a local DHT of 1000 peers.

Corel: For the second data set we experimented on $60,000$ photo images from the Corel data set as previously used in, e.g. [ORC+98] [1]. Each image has 89 dimensions in this data set. In this case we assumed a global DHT of $100,000$ peers and 100 peers per local DHT.

For both datasets, we use the Euclidean distance to measure the distances between points, treating all dimensions equally and without preprocessing the data. As query points we chose 100 points randomly from each of the datasets. All performance measures are averaged over 100 queries. $K = 20$ in all KNN experiments. Table 6.1 summarizes the data set and overlay setup parameters.

Methods under Comparison

To evaluate our data placement methods, we distribute the data among peers once with ξ_{sum} and once with ξ_{lsh}. We experimented with different values of LSH parameters: k, W and W_2 and here report the best performances achieved. In the results, unless otherwise stated, the default values are $k = 20$, $W = 5$ and $W_2 = 3$ for the Flickr data set and $k = 20$, $W = 50$ and $W_2 = 1.25$ for the Corel data set. For each of these mapping functions we consider the following query processing methods:

Simple: This is the baseline query processing algorithm. At query time, the whole local index of the peer which is responsible for the mapped LSH bucket using ξ is scanned without further forwarding. This is used both for KNN and range queries.

MProbe: At KNN query time we use the multi-probing based algorithm as described in Section 6.4.2, fixing the number of probes to 100.

Linear: At query time the linear forwarding algorithm, Section 6.4.1, is used with appropriate stopping conditions for KNN or range search.

Sample: This is the sampling-based method described in Section 6.5.1 which is dedicated to range query processing.

[1] available under:
http://kdd.ics.uci.edu/databases/CorelFeatures/CorelFeatures

Sahin: To compare with state of the art, we have implemented the method described in Section 3.3.1 by Sahin et. el [SEAA04]. In order to fairly compare against this method, we follow the same index creation of Section 6.3 where replicas of the data are maintained by smaller rings. We have experimented with different number of reference vector sizes and different number of references for publishing indices. We report here the best performance results which achieve a fair load balance as well. The reference vector size is set to 32 and reference points are selected uniformly at random from the whole data set. To achieve a fair load balance, we employed multi-level reference sets as described in the initial work, however increasing the number of references used for publishing the indices proved to be more effective. Therefore only one level of references is used and the number of references used for publishing indices is set to 4. The initial work of [SEAA04] employs the *Simple* query processing method as explained in 3.3.1. In addition to that we also experimented processing queries with our *Linear* method. The *MProbe* method is not applicable here, as it depends on the LSH buckets. Processing range queries are not discussed in [SEAA04]; we also experimented only the KNN queries with this method.

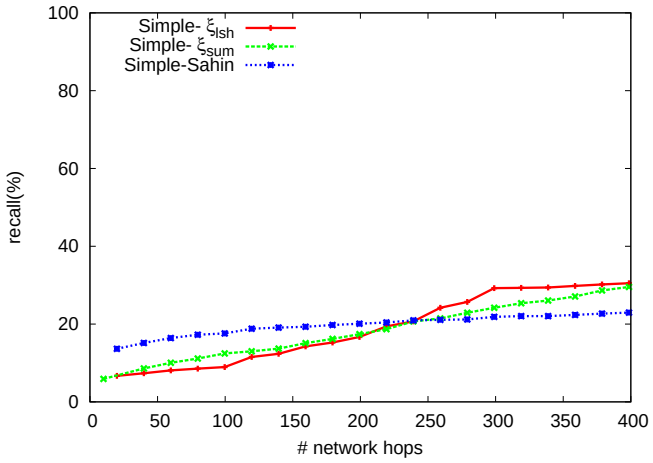


Figure 6.3: Recall versus number of DHT lookups for different data placement methods employing *Simple* query processing for the Flickr data set

Data set	Sahin	ξ_{sum}	ξ_{lsh}
Flickr	0.42	0.52	0.41
Corel	0.40	0.46	0.57

Table 6.2: Gini Coefficient when distributing 2 replicas of the data sets

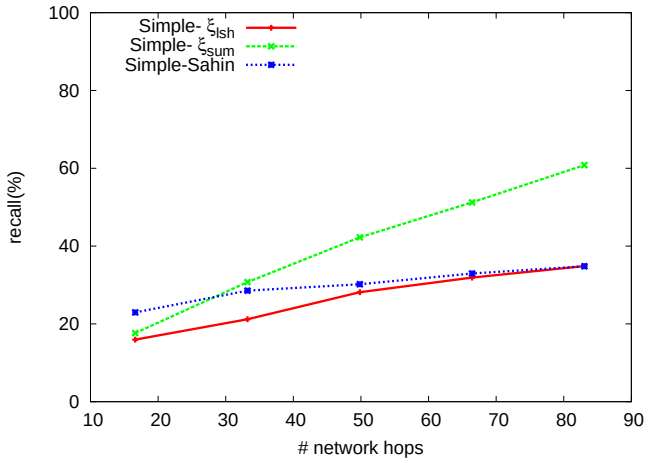


Figure 6.4: Recall versus number of DHT lookups for different data placement methods employing *Simple* query processing for the Corel data set

Measures of Interest

Gini Coefficient: As for a measure of load imbalances we consider the Gini coefficient of the load distribution, that is defined as $G = 1 - 2\int_0^1 L(x)dx$ where $L(x)$ is the Lorenz curve of the underlying distribution. Pitoura et al [PT07] show that the Gini coefficient is the most appropriate statistical metric for measuring load distribution fairness. The Gini coefficient, apart from the other three measures, is query independent and measured once for each benchmark to report on the storage load distribution.

Number of Network Hops: We count the number of network hops during the query execution. Network hops are one of the most critical parameters in making distributed algorithms applicable in large-scale wide-area networks. Each DHT lookup causes $log n/2$ or $log N/2$ network hops (i.e., local or global DHT). Hence, we count the number of local and global lookups and translate this to the overall number of network hops. The cost for local query execution is considered to be negligible in our scenario, as the network cost is clearly the dominating factor: One single network hop in a wide-area costs in average around $100ms$, which overrules the I/O cost, induced by a standard hard disk, with approximately $8ms$ for disk seek time plus rotation latency and $100MB/s$ transfer rate for sequential accesses, in case of local disk access.

Relative Recall: For the effectiveness metric, we report on the relative recall, i.e., the number of relevant data points among returned data points. The relevance is defined by the full-scan run over the entire data set to determine

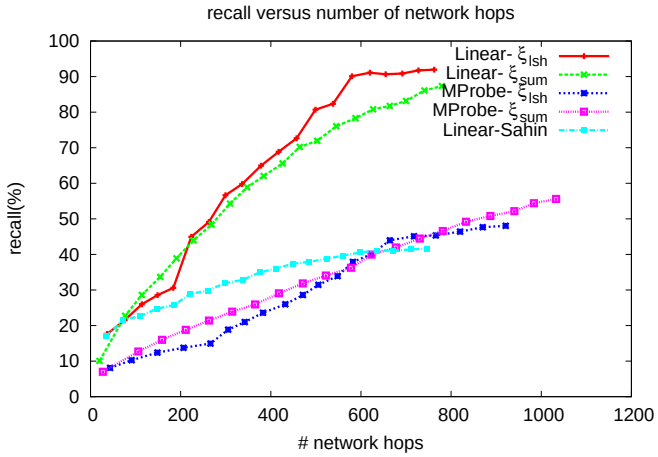


Figure 6.5: Recall versus number of DHT lookups for the Flickr data set representing *Linear* with the three different placement methods and *MProbe* with ξ_{sum} and ξ_{lsh}

the K nearest points to a query point for KNN queries. For range queries, all data points in range r of the query point are relevant. It should be noted that since we are ranking all candidate objects and returning only the top K in KNN queries and only the points within distance r of the query point in range queries, precision is equal to relative recall and we do report it separately.

Error Ratio: Given that LSH is an approximate algorithm, we also measured the *Error Ratio* which measures the quality of approximate nearest neighbor search as defined in [GIM99]. $\frac{1}{K}\sum_{i=1}^{K}\frac{d_{LSH_i}}{d_{true_i}}$ where d_{LSH_i} is the distance of query point to its i-th nearest neighbor found by LSH and d_{true_i} is its distance to its true i-th nearest neighbor. Since this measure does not add new insight over relative recall and due to space constraints we do not report it here.

6.6.2 Experimental Results

We first investigate the effect of employing ξ_{sum} and ξ_{lsh} on the load distribution and compare this against the *Sahin* data placement. As seen in Table 6.2 for both Flickr and Corel data sets, the Gini coefficients of all different load distributions fall in the range of $[0.4, 0.6]$ which is a strong indicator of a fair load distribution [PNT06].

KNN query Results

We now show the results obtained for the KNN search. Figures 6.3 and 6.4 show the obtained recall when queries are processed by the *Simple* method. We have varied the number of hash-tables (or respectively replicas for *Sahin*) from

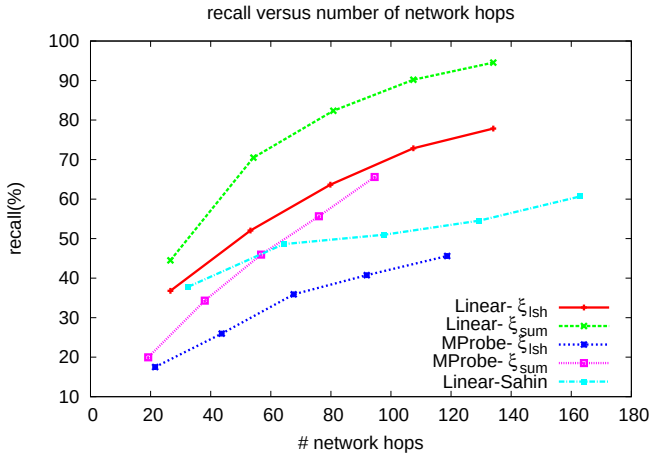


Figure 6.6: Recall versus number of DHT lookups for the Corel data set representing *Linear* with the three different placement methods and *MProbe* with ξ_{sum} and ξ_{lsh}

2 to 40 for the Flickr data set and from 2 to 10 for the Corel data set. For the Corel dataset ξ_{sum} achieves better recall compared to *Sahin* and ξ_{lsh}. ξ_{sum} and ξ_{lsh} obtain better recalls for number of replicas more than 24 for the Flickr data set which has higher dimensionality . We observe that the obtained recall for all three placement methods with the *Simple* query processing algorithm is quite low even when increasing the number of hash-tables (replicas). It should be noted that while employing the *Simple* method for processing queries, the number of incurred network hops for answering each query is equal to number of hash-tables (replicas) times $logN/2$, where N is the number of peers in the global DHT. In this case, only one peer is visited in each local DHT maintaining a hash-table (replica) of the data set.

Figures 6.5 and 6.6 show recall versus number of network hops for three placement methods, this time using *Linear* and *MProbe* processing algorithms respectively for the Flickr and Corel data sets. We see a big increase in recall compared to when processing queries using *Simple* for both data sets and all three placement methods. We have varied the number of hash-tables (replicas) exactly like the previous experiment, from 2 to 40 for Flickr, and from 2 to 10 for Corel. We show on the x-axis the number of network hops incurred which corresponds to the number of hash-tables (replicase) and number of times the query is forwarded in each local DHT maintaining a hash-table (replica). As can be seen for both data sets, the combination of ξ_{sum} and ξ_{lsh} with the *Linear* processing algorithm achieves the best recall while incurring not many network hops. This confirms that ξ_{sum} and ξ_{lsh} preserve the locality, i.e., they

		Relative Recall in % (#Network Hops)		
placement	**l**	**Simple**	**MProbe**	**Linear**
Sahin	2	13.65% (19)	-	17.00% (34)
ξ_{sum}	2	6.90%(19)	8.50%(27)	**18.65%(19)**
ξ_{lsh}	2	6.70% (19)	8.10% (43)	17.65% (36)
Sahin	10	17.60% (99)	-	25.80% (185)
ξ_{sum}	10	12.45% (99)	21.40% (262)	**38.90% (190)**
ξ_{lsh}	10	8.95% (99)	14.95% (266)	30.60% (183)
Sahin	20	20.10% (199)	-	34.95% (375)
ξ_{sum}	20	17.40% (199)	34.05% (522)	62.05% (384)
ξ_{lsh}	20	16.70% (199)	28.60% (471)	**64.95% (378)**
Sahin	30	21.85% (298)	-	39.70% (559)
ξ_{sum}	30	24.20% (298)	46.60% (781)	78.30% (587)
ξ_{lsh}	30	29.25% (298)	43.95% (664)	**90.10% (579)**
Sahin	40	22.95% (398)	-	41.65% (746)
ξ_{sum}	40	29.55% (398)	55.60% (1032)	87.35% (779)
ξ_{lsh}	40	30.50% (398)	48.05% (921)	**91.95% (762)**

Table 6.3: Measuring recall and number of network hops for different number of hash-tables, for different placement and processing methods the Flickr data set.

		Relative Recall in % (#Network Hops)		
placement	**l**	**Simple**	**MProbe**	**Linear**
Sahin	2	22.95% (16)	-	37.85% (32)
ξ_{sum}	2	17.65% (16)	19.95% (19)	**44.49% (26)**
ξ_{lsh}	2	15.95% (16)	17.50% (21)	36.80% (26)
Sahin	10	34.80% (83)	-	60.70% (163)
ξ_{sum}	10	60.79% (83)	65.35% (94)	**94.55% (134)**
ξ_{lsh}	10	34.85% (83)	45.60% (118)	77.84% (134)

Table 6.4: Measuring recall and number of network hops for different number of hash-tables, for different placement and processing methods the Corel data set.

group buckets with similar content to the same or neighboring peers. ξ_{sum} and ξ_{lsh} achieve similar recall in both data sets, while *Sahin*'s quality of result degrade drastically when processing the Flickr data set. This observation again shows better scalability of our algorithm with respect to data dimensionality, which is due to LSH characteristics. *MProbe* achieves better recall compared to *Simple*, but does not perform as well as *Linear*: number of incurred network hops is more, as each forward in a local DHT maintaining a hash-table results in $logn/2$ hops, where n is the number of peers maintaining that ring. We have also summarized these results in Tables 6.3 and 6.4 to better compare these methods with respect to network load (number of times the data is replicated in the network). For example let us consider the 13^{th} and 15^{th} rows of Table 6.3. With *Linear* the ξ_{lsh} data placement achieves more than twice better recall at the expense of only 0.01 more number of network hops while having the same network load as *Sahin*. The best achieved recall is shown in bold in both tables.

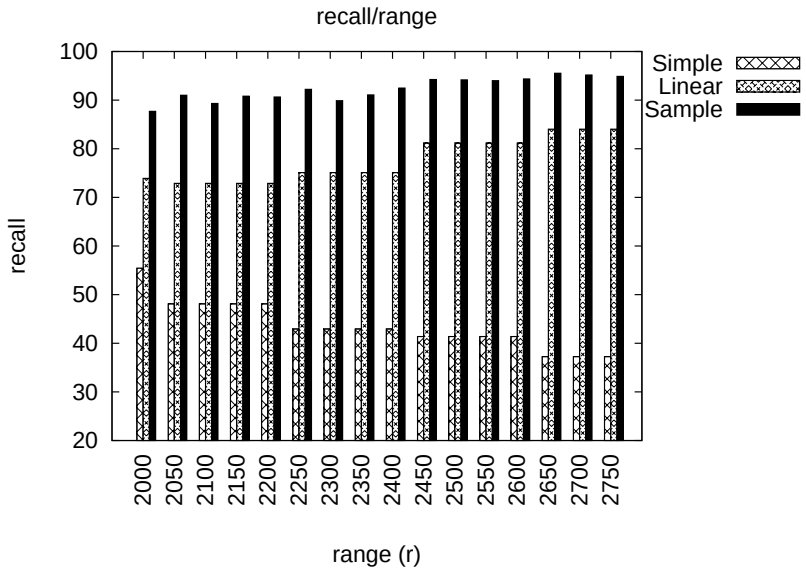


Figure 6.7: The effect of varying the range on recall. The results are shown for #hash-tables=20 and ξ_{sum} as the placement function for the Corel data set

Range query Results

We now report the results obtained for range searches. For each data set the radius of the range query is chosen such that the number of possible results are reasonable, i.e. for Flickr this ranges from 27 to 562, while for Corel it ranges from 17 to 194. Figure 6.7 shows a general view of the effect of varying the range on recall. As discussed also in Section 6.5.1 recall in the *Simple* method can fall when the radius of range increases. An example of this effect is shown in Figure 6.7. However, our Sampling method proves to be effective at maintaining high recall as the radius changes. Tables 6.5 and 6.7 show the results for ξ_{sum}, while Tables 6.6 and 6.8 report on ξ_{lsh} for the two data sets. Clearly, *Sample* performs very well at achieving good recall for different choices of the range in both data sets. Our *Linear* method obtains smaller recall compared to *Sample*, however the number of network hobs incurred by this method is considerably less than *Sample*. The best achieved recalls are shown in bold.

6.7 Conclusions

We present a robust and scalable solution to the distributed similarity search problem over high dimensional data. Having investigated the characteristics of the existing centralized LSH based methods, we devise an algorithm to distribute

		Relative Recall in % (#Network Hops)		
range	**l**	**Simple**	**Linear**	**Sample**
2000	2	60.98% (17)	77.55% (24)	**80.09% (41)**
2000	10	81.04% (83)	95.95% (119)	**95.00% (200)**
2150	2	49.48% (17)	70.17% (25)	**75.26% (43)**
2150	10	74.10% (83)	96.01% (124)	**95.47% (211)**
2300	2	42.68% (17)	72.10% (26)	**78.02% (47)**
2300	10	68.87% (83)	95.82% (130)	**96.54% (224)**

Table 6.5: Measuring recall and number of network hops for different range radius' and number of hash-tables, for different processing methods under comparison with ξ_{sum} as the placement function for the Corel data set.

		Relative Recall in % (#Network Hops)		
range	**l**	**Simple**	**Linear**	**Sample**
2000	2	51.88% (17)	63.22% (24)	**70.73% (42)**
2000	10	66.95% (83)	82.86% (116)	**93.16% (210)**
2150	2	40.05% (17)	54.03% (24)	**61.52% (44)**
2150	10	54.31% (83)	77.98% (120)	**91.99% (222)**
2300	2	33.23% (17)	50.39% (25)	**61.40% (48)**
2300	10	49.36% (83)	76.46% (124)	**94.13% (239)**

Table 6.6: Measuring recall and number of network hops for different range radius' and number of hash-tables, for different processing methods under comparison with ξ_{lsh} as the placement function for the Corel data set.

the p-stable LSH method considering the requirements that arise in distributed settings. Our proposed locality preserving mapping, brings together two *contradictory* conditions of efficient and high quality similarity search in distributed se1ttings: Enabling probabilistic placement of similar data on the same peer or neighboring peers, while achieving a fair load balancing. We describe the process of creating the index, leveraging our proposed mapping and its characteristics. We theoretically prove the locality preserving properties of our mapping and devised efficient algorithms for both K-nearest neighbor and range queries. To our knowledge this is the first work enabling similarity range queries over LSH indices. Our experimental evaluation shows major performance gains compared to state-of-the-art. We believe that our approach is thus well-positioned to become a fundamental building block towards applying LSH based methods in real world, distributed applications.

		Relative Recall in % (#Network Hops)		
range	**l**	**Simple**	**Linear**	**Sample**
200	2	79.38% (20)	80.67% (26)	**81.32% (53)**
200	10	80.82% (100)	85.00% (138)	**86.54% (264)**
200	20	82.02% (199)	87.06% (275)	**89.98% (530)**
200	30	84.03% (299)	89.96% (414)	**93.04% (795)**
200	40	86.45% (399)	92.92% (553)	**95.88% (1060)**
225	2	61.82% (20)	63.14% (27)	**64.02% (54)**
225	10	64.85% (100)	70.44% (141)	**72.75% (269)**
225	20	67.82% (199)	75.42% (282)	**79.09% (540)**
225	30	70.66% (299)	80.70% (423)	**85.63% (812)**
225	40	74.29% (399)	84.85% (566)	**89.99% (1086)**
250	2	41.63% (20)	44.10% (29)	**45.45% (56)**
250	10	44.58% (100)	53.51% (149)	**57.45% (282)**
250	20	47.69% (199)	60.30% (296)	**66.15% (567)**
250	30	51.26% (299)	66.95% (449)	**73.89% (857)**
250	40	55.20% (399)	72.73% (600)	**80.16% (1147)**
275	2	22.00% (20)	26.36% (34)	**28.70% (66)**
275	10	25.59% (100)	40.21% (177)	**43.99% (330)**
275	20	29.81% (199)	49.33% (351)	**55.87% (659)**
275	30	33.77% (299)	58.03% (530)	**67.12% (999)**
275	40	37.47% (399)	64.36% (707)	**74.62% (1335)**
300	2	11.80% (20)	18.94% (46)	**22.98% (89)**
300	10	14.92% (100)	36.50% (234)	**43.14% (434)**
300	20	18.85% (199)	48.42% (466)	**57.06% (870)**
300	30	23.16% (299)	59.27% (707)	**68.77% (1329)**
300	40	27.55% (399)	66.62% (947)	**76.55% (1784)**

Table 6.7: Measuring recall and number of network hops for different range radius' and number of hash-tables, for different processing methods under comparison with ξ_{sum} as the placement function for the Flickr data set.

		Relative Recall in % (#Network Hops)		
range	**l**	**Simple**	**Linear**	**Sample**
200	2	78.90% (20)	81.44% (27)	**81.49% (51)**
200	10	79.66% (100)	84.08% (134)	**84.58% (256)**
200	20	81.06% (199)	86.89% (270)	**88.10% (518)**
200	30	84.75% (299)	90.59% (409)	**92.54% (780)**
200	40	85.89% (399)	92.07% (542)	**93.92% (1034)**
225	2	61.31% (20)	64.55% (28)	**65.34% (53)**
225	10	61.94% (100)	67.05% (136)	**68.82% (261)**
225	20	64.63% (199)	73.50% (276)	**76.44% (531)**
225	30	69.53% (299)	80.54% (418)	**84.36% (800)**
225	40	70.41% (399)	82.75% (552)	**86.98% (1058)**
250	2	41.59% (20)	47.53% (31)	**48.36% (57)**
250	10	42.62% (100)	51.36% (144)	**54.56% (272)**
250	20	45.43% (199)	59.17% (294)	**65.20% (558)**
250	30	52.92% (299)	69.43% (448)	**78.37% (845)**
250	40	54.72% (399)	72.25% (586)	**80.91% (1112)**
275	2	22.45% (20)	29.58% (36)	**31.82% (65)**
275	10	24.26% (100)	35.38% (159)	**40.07% (300)**
275	20	27.09% (199)	45.42% (335)	**51.63% (628)**
275	30	35.82% (299)	62.10% (522)	**71.02% (970)**
275	40	37.76% (399)	64.63% (670)	**73.61% (1256)**
300	2	11.61% (20)	21.90% (46)	**24.41% (80)**
300	10	13.46% (100)	31.17% (194)	**35.61% (364)**
300	20	17.22% (199)	45.34% (420)	**52.43% (791)**
300	30	25.04% (299)	64.39% (671)	**73.05% (1235)**
300	40	27.22% (399)	67.00% (846)	**76.10% (1574)**

Table 6.8: Measuring recall and number of network hops for different range radius' and number of hash-tables, for different processing methods under comparison with ξ_{lsh} as the placement function for the Flickr data set.

Chapter 7

Conclusion

Ranking queries have received a lot of attention in the past few years. Processing this kind of queries in recently emergent environments, however, poses new challenges which were non existing previously. More specifically, the requirements of an efficient ranking query operator in a data stream model or a distributed P2P setting are different from those of a traditional, centralized, fully under control database system. Throughout this thesis, we have developed novel methods for evaluating ranking queries in these new settings, which have not been considered before.

Due to its wide spread in today's data-centric applications, we look into the data stream model as our first non-conventional setting. We investigate processing top-k queries over multiple non-synchronized streams in a sliding window model. The exact score of an incoming object with regard to a top-k query registered at the system, can not be calculated instantly, due to different attributes of it arriving in different streams which are not in sync. In such a setting where the data streams have very high incoming rate and in face of limited available main memory, it is infeasible to store *all* incoming tuples. Therefore, it is desirable to retain only the interesting ones, which are identified according to the query aggregation function. We propose an exact scheme based on multiple instance creations which enables the system to drop only those objects which will never be a result of the top-k query during their life-time. We further show that the memory usage is still growing linearly with the window size. We propose an approximate method which leverages inter-stream statistics to better estimate the score of incomplete objects. With this scheme we are able to limit the storage dramatically with only minor losses in the accuracy of results.

Following our work on top-k processing over data streams, we identify a novel application which relies on processing large number of top-k queries in real-time over a stream of textual data. With the popularity of Web 2.0 portals, it is difficult for human users to keep up-to-date to the interesting events, such as new weblog posts, or twitter status messages. In order to assist the user in staying tuned to this ocean of new information, and not drowning by spending

too much time and energy in finding what is of interest for them, we consider a system where users can subscribe with their profiles (as set of keyword) and will be updated with only the top-k related events of their interest periodically. The main differences between this work and the previous problem are the dimensionality of the data, as well as the cardinality of the subscribed queries. In such a setting, indexing the queries is beneficial, as otherwise all queries have to be evaluated with an incoming document. We design a query index, similar to the traditional inverted index, but different in the scores used to order the lists. Our proposed scheme enables early stopping, which is crucial for the system to keep up with the high rate of incoming data, without loosing the properties of being real-time. As keeping the lists completely sorted incurs high costs, we propose another scheme based on retaining the lists sorted only partially. Our scheme proves to be effective when the boundaries of the lists are chosen with regard to our defined cost model. Furthermore, we propose a scheme for result maintenance of each query, to avoid the problem mentioned above when keeping the dominant set over a large number of points. The combination of our partially ordered list scheme with the result maintenance method proves to be very effective in decreasing the overall processing time.

We move to ranking queries over a distributed P2P setting as our next non-conventional setting. Here we consider the problem of k nearest neighbor query processing over high dimensional data, where the data is distributed among peers of a P2P overlay network. Our solution relies on the so-called family of locality sensitive hashing functions and provides approximate results. Our solution satisfies the two requirements of an appropriate mapping of data to the peer space: it probabilistically keeps similar objects on the same peer or neighboring peers and ensures a fair load balance over the network.

7.1 Impact and Future Work

In previous streaming models, the unit of data is usually one incoming data tuple. Our model of processing *objects*, however, takes a broader view on the entities under process, as a consequence of which incomplete objects come in to the picture when several non-synchronized sources of information report on objects. We provide fundamental results on the space limitations of ranking queries over incomplete data streams and propose an approximate solution which exploits inter-stream correlations. Our work on processing ranking queries under this model is a first step in filling out the required blocks to better understand and develop this model. It motivates many other questions to be considered in this recurring, but novel model of incomplete data streams. Considering other classes of queries, identifying the difficulties which surface up in processing them and designing algorithms which address the challenges under this model are necessary to produce a system for data streams complying with this model.

In this thesis we consider centralized processing of dynamic data, or, distributed processing in case of static data. Sources of dynamic data are distributed in many streaming scenarios, such as in sensor networks or in the

context of Web 2.0 streams. Therefore, it is natural to consider pushing some portions of the computations to the producing nodes to exploit the power of distributed computing and also effectively reduce the rate of incoming data to the centralized server. However, distributed processing of dynamic data can bring about new challenges. Some data sources have very limited computational resources, (e.g., cheap sensors) which call for efficient algorithms that can be run over resource limited machines. On the other hand, distributing holistic aggregation functions, which require access to the whole data to provide results, needs extra care. Problems such as these can be a subject of future work. Some further refined problems are described below.

In the area of continuous top-k processing over data streams in a sliding window model:
The solution described for continuous top-k processing in Chapter 4 uses straight forward structures to maintain the dominance set of a dynamically changing set of points. Designing tailored data structures for this problem which can improve the performance of the system significantly is left as future work. Also, as our work in Chapter 5 illustrates, maintaining the top part of the dominant set, instead of maintaining the whole set, can have great performance gains. Exploring these possibilities remain as part of our future work. Furthermore, we did not present an online algorithm for when only fixed amount of memory is available. To be able to design appropriate online solutions for the mentioned problem, the streaming data should be modeled first. We refer the interested reader to similar work in modeling streams for approximate join processing under fixed memory assumptions in [SW04].

In the area of k nearest neighbor queries over P2P networks:
Although our solution is targeted for a distributed setting, we map the high dimensional data to the *one* dimensional peer space. This solution can be promising also in centralized search where the data is stored on local disk or in clustered computers, where deriving statistics over the data is easier, therefore tuning the parameters of the hash function is easier. Conventional indices, such as a B+ tree could be utilized to index the one dimensional buckets. Comparison with the recent work in [TYSK09] which also maps the LSH output to one-dimension is a good stating point.

Appendix A

Proofs

Proof of Theorem 1 We first show that if $p^i \notin S_k$ then p^i cannot be a top-k result. If $p^i \notin S_k$ then there are at least k distinct instances which dominate p^i. From the interval dominance definition it follows that at least k distinct instances exist which have higher *currentscore* than the best score p^i can ever get and live longer than p^i. Since dominance is persistence, p^i cannot be part of top-k anytime during its life time, as there are at least k preferred instances all its life. For showing the necessity of keeping S_k, assume $p^i \in S_k$. We describe a case where p^i is part of top-k results. There are at most $k-1$ other distinct objects which live longer than p^i and their *currentscore* is higher than p^i.*bestscore*. Let τ be the time when all instances which have better scores than p^i and are older, expire. Assume no new tuples arrive at any of the streams until τ. p^i will be a top-k result at τ. □

Proof of Lemma 1 Let $X_1, X_2, ..., X_n$ be iid random variables following a probability distribution function $f(x)$, denoting the n *currentscore*'s we are considering. Note that the iid assumption is valid, since we are not considering several instances of the same object, but objects with distinct *id*s. We can re-order them

$$S_1 \leq S_2 \leq ... \leq S_n$$

where S_κ is called the κ-th order statistic. We are interested in the maximum, which is the S_n-th order statistic. Since the difference between $currentscore_i$ and $bestcore_i$ is at most ϵ, points which are not dominated by any other point are those whose *bestscore* is at least $max_{1 \leq i \leq n} currentscore_i$ which can be estimated by $E[S_n]$. Hence, we have the following lower bound for the expected size of the dominance set:

$$E[|S^*|] \geq n * \int_{E[S_n]-\epsilon}^{E[S_n]} f(x)dx$$

This is a lower bound as some objects whose *bestscore* is smaller than S_n are part of the dominance set due to their *time* attribute. In case of standard

uniform distribution $f(x) = 1$ and $E[|S^*|] \geq n*\epsilon$ and is independent of $E[S_n]$. □

Proof of Theorem 2 For simplicity, we give the proof for two streams and $k = 1$. We present an input distribution which satisfies our bound $W - m$.

Let *online* denote any online strategy. Assume we have received m tuples from stream s_1 where all tuples have equal values v_1. The memory is filled with the distinct m instances which corresponds to these tuples, all having equal scores $f(v_1, 0)$. Assume a new tuple, with the same value arrives at s_1. The memory is full so one of the instances should be evicted. Let p^1 denote the instance which is removed by *online*. Now a tuple, $< p.id, v_2, p.t_2 >$ arrives in stream s_2 where $v_2 < v_1$. If we didn't have fixed memory, instance p^1 would be the top-1 object, as $score(p^1) = f(v_1, v_2)$ which is larger than $f(v_1, 0)$ due to monotonicity of f. From this point on, all objects arriving in s_1 or s_2 are distinct and have values smaller than v_1 and larger than v_2. Therefore for at least $W - m$ timestamps the true top-1 object will be p^1. However since *online* had evicted p^1, it will report some other object as the top-1 result. Note that *online* does not report p^2, which would have the same id as p^1 since $score(p^2) = f(0, v_2) < f(v_1, 0)$. OPT however, knows which tuples are arriving so it would not have evicted p^1. Therefore, for $W - m$ timestamps OPT has precision 1 while online has precision 0. □

Proof of Theorem 3 We show that for any profile which has not been updated before the stopping condition is reached, d does not serve as a top-k result. In other words we show that for such profiles, $sim(d, p) < p.s$. If p has been seen in one of the sorted lists before the stopping condition, according to the algorithm its similarity score with d has been evaluated by looking up p in the profile hash-table. Therefore if p has not been updated, clearly $sim(d, p) < p.s$. Now assume p has not been observed in any of the sorted lists before the stopping condition. For a list l_i let $\underline{v_i}$ be the last observed value under sorted access. Since the lists are sorted in descending values, $p.v_i < \underline{v_i}$. As a result of this and due to f and g's monotonicity, $g(f_{w_1}(v_1), ...f_{w_m}(v_m)) \leq g(f_{w_1}(\underline{v_1}), ...f_{w_m}(\underline{v_m})) < 1$ where the last equality is the stopping criteria. Since $p.v_i = p.u_i/p.s$ and due to f and g's homogeneity, we have

$$g(f_{w_1}(v_1), ...f_{w_m}(v_m)) = g(f_{w_1}(u_1)/p.s, ...f_{w_m}(u_m)/p.s)$$
$$= g(f_{w_1}(u_1), ...f_{w_m}(u_m))/p.s < 1$$

which is equivalent to $sim(d, p) < p.s$. □

Proof of Theorem 4 We first show that if $\tau \leq p.R.score$, $p.R$ contains the true top-k. Let d be the valid document with the largest score which is not in $p.R$ at current time $t_{current}$ and $p.R.score_k$ be the score of the ranked k document in $p.R$ also at $t_{current}$. We show that $sim(d, p) < R.score_k$. We should consider two cases: first d was inserted to $p.R$ but then removed, or d was never inserted to $p.R$. Since d is valid, it was removed from $p.R$ as a result of being dominated by k documents which means k documents with

longer life times exist which have a higher score than d. These are indeed in $p.R$, as we have assumed d has the largest score among all valid documents *not* in $p.R$. So for the first case $sim(d,p) < p.R.score_k e$. In the second case, d was never inserted to $p.R$. Let t_1 denote the time when the most recent re-evaluation was performed. If $t_1 > d.time$, the most recent re-evaluation was performed after d's arrival. Since d was not inserted in $p.R$, at least k documents with higher scores than d existed at time t_1. Since $|p.R| = k$ after each re-evaluation, τ at after t_1 and before a new re-evaluation is equal to or larger than the score of the ranked k document at time t_1. Since we have assumed the most recent re-evaluation happened in t_1, either those top-k documents have not expired until $t_{current}$, or documents with score larger than τ_{t_1} have arrived, otherwise the size of $p.R$ would be less than k at some point after t_1 which is in contradiction with our assumption that the most recent re-evaluation was invoked at t_1. In both cases $R.score_k \leq \tau_{t_1} > sim(d,p)$. Now assume $t_1 < d.time$: the most recent re-evaluation happened before d's arrival. In this case, $sim(d,p) < \tau_{d.time}$, otherwise d was inserted in $p.R$. If no re-evaluations happens, τ can only increase, as only higher scored documents can be inserted to $p.R$. Similar to the previous case, either documents in $p.R$ at time $d.time$ have not expired yet or higher scored documents have arrived, otherwise a re-evaluation would have been fired. In both cases, $R.score_k \geq \tau_{d.time} > sim(d,p)$ which completes the proof for correctness of results when $\tau \leq R.score$.

To show that this is also a necessary condition, we give an example of when $p.R$ does not contain top-k results if $\tau < R.score$. For simplicity let $k = 2$, examples for other k can be constructed similarly. Let $\tau = R.score + \epsilon$ and $\epsilon > 0$. Assume R is empty and consider the following stream of documents (first attribute shows time and the second is score with regard to the specific profile we consider): $d_1(1, s_1)$, $d_2(2, s_2)$, $d_3(3, s_3)$, $d_4(4, s_4)$, where $s_1 > s_2$, $s_1 > s_3$, $s_3 > s_2$. Then when d_4 arrives, $\tau = s_2 + \epsilon$, because d_2 is dominated only by d_3 so it isn't removed. Now if $s_4 = s_2 + \epsilon/2$, d_4 is not inserted to R. Assume no new document arrives. When d_1 expires, d_2 and d_3 are reported as the top-k results although d_4 has higher score than d_2. □

Proof of Theorem 5 Let $s(c,\delta) := pr(|h(\mathbf{q}) - h(\mathbf{v})| \leq \delta)$ and $t(c,\delta) := pr(|h(\mathbf{q}) - h(\mathbf{v})| = \delta)$ where $||\mathbf{q} - \mathbf{v}||_2 = c$. Then $s(c,\delta)$ can be written as $s(c,\delta) = t(c,0) + \cdots + t(c,\delta)$. We want to show that for any fixed δ, $s(c,\delta)$ is monotonically decreasing in terms of c. We first derive $t(c,\delta)$. Our argument is similar to that of [DIIM04]. Since elements of the random vector $\mathbf{a}$ are chosen from a standard Normal distribution, $\mathbf{a}.\mathbf{q} - \mathbf{a}.\mathbf{v}$ is distributed according to cX where X is a random variable drawn from a Normal distribution. Therefore the probability distribution of $|\mathbf{a}.\mathbf{q} - \mathbf{a}.\mathbf{v}|$ is $\frac{1}{c} f(\frac{x}{c})$ where $f(x)$ denotes the probability density function of the absolute value of the standard Normal distribution (i.e., the mean is zero and the variance is one) :

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ \frac{2}{\sqrt{2\pi}} e^{-x^2/2} & \text{if } x \geq 0 \end{cases}$$

For $\delta = 0$, in order to have $|h(\mathbf{q}) - h(\mathbf{v_1})| = \delta$, $|\mathbf{a}.\mathbf{q} - \mathbf{a}.\mathbf{v}|$ has to be in $[0, W)$. Depending on the exact value of $|\mathbf{a}.\mathbf{q} - \mathbf{a}.\mathbf{v}|$ different range of values for B can leave $|h(\mathbf{q}) - h(\mathbf{v_1})| = 0$ or change it to $|h(\mathbf{q}) - h(\mathbf{v_1})| = 1$. For example if $|\mathbf{a}.\mathbf{q} - \mathbf{a}.\mathbf{v}| = 0$, all values of $B \in [0, W)$ keep the desired values $|h(\mathbf{q}) - h(\mathbf{v_1})| = 0$. The size of this range of values for B decreases linearly as $|\mathbf{a}.\mathbf{q} - \mathbf{a}.\mathbf{v}|$ increases inside $[0, W)$, until it reaches 0 for $|h(\mathbf{q}) - h(\mathbf{v_1})| = W$. Since B is drawn uniformly at random from $[0, W]$, the following can be derived:

$$t(c, 0) = \int_0^W \frac{1}{c} f(\frac{u}{c})(1 - \frac{u}{W})du$$

The case for $\delta > 0$ is similar. However this time, a bigger range of values ($[(\delta - 1)W, (\delta + 1)W)$) for $|\mathbf{a}.\mathbf{q} - \mathbf{a}.\mathbf{v}|$ can satisfy $|h(\mathbf{q}) - h(\mathbf{v_1})| = \delta$. The argument regarding B is similar to above. Therefore the following can be seen:

$$t(c, \delta) = \int_{(\delta-1)W}^{(\delta)W} \frac{1}{c} f(\frac{u}{c})(\frac{u}{W} - (\delta - 1))du$$

$$+ \int_{(\delta)W}^{(\delta+1)W} \frac{1}{c} f(\frac{u}{c})((\delta + 1) - \frac{u}{W})du \tag{A.1}$$

Summing up all values of $t(c, d)$ for $d \leq \delta$, we arrive at the following for $s(c, \delta)$:

$$s(c, \delta) = \int_0^{(\delta+1)W} \frac{1}{c} f(\frac{u}{c})du + \int_{(\delta)W}^{(\delta+1)W} \frac{1}{c} f(\frac{u}{c})(\delta - \frac{u}{W})du$$

With a change of variable $v = \frac{u}{c}$ we can eliminate all occurrences of c inside the integrals and take the derivative of $s(c, \delta)$ in terms of c. Given that $\int uf(u)du = -f(u)$, this will lead us to:

$$\frac{\partial s(c, \delta)}{\partial c} = \frac{1}{W}(f(\frac{(\delta + 1)W}{c}) - f(\frac{(\delta)W}{c}))$$

which is smaller than 0 for all values of $c > 0$, as $f(x)$ is monotonically decreasing. Therefore for any fixed δ, $s(c, \delta)$ is monotonically decreasing in terms of c. □

Proof of Theorem 6 Let $||\mathbf{q} - \mathbf{v}||_2 = c$. From Theorem 5, $pr(|h(\mathbf{q}) - h(\mathbf{v})| = \delta)$ is equal to Eq. A.1. It is easy to see that if we take the derivative from this equation in terms of δ we arrive at the following:

$$\frac{\partial t(c, \delta)}{\partial \delta} = -\int_{\frac{(\delta-1)W}{c}}^{\frac{(\delta)W}{c}} f(u)du + \int_{\frac{(\delta)W}{c}}^{\frac{(\delta+1)W}{c}} f(u)du$$

which is smaller than zero, as the range of the two integrals is equal, the term under both is the same and is monotonically decreasing and non negative for the values under comparison. □

Bibliography

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[Abe01] Karl Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, pages 179–194, 2001.

[ABN06] Ittai Abraham, Yair Bartal, and Ofer Neiman. Advances in metric embedding theory. In *STOC*, pages 271–286, 2006.

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. Cql: A language for continuous queries over streams and relations. In Georg Lausen and Dan Suciu, editors, *DBPL*, volume 2921 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2003.

[ACc+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, C. Erwin, Eduardo F. Galvez, M. Hatoun, Anurag Maskey, Alex Rasin, A. Singer, Michael Stonebraker, Nesime Tatbul, Ying Xing, R. Yan, and Stanley B. Zdonik. Aurora: A data stream management system. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *SIGMOD Conference*, page 666. ACM, 2003.

[AGMS02] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. *J. Comput. Syst. Sci.*, 64(3):719–747, 2002.

[AI06] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.

[APPK08] Vassilis Athitsos, Michalis Potamias, Panagiotis Papapetrou, and George Kollios. Nearest neighbor retrieval using distance-based hashing. In *ICDE*, pages 327–336, 2008.

[AY08] Charu C. Aggarwal and Philip S. Yu. A framework for clustering uncertain data streams. In *ICDE*, pages 150–159, 2008.

[BAS04] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 353–366, New York, NY, USA, 2004. ACM Press.

[BB05] Erik Buchmann and Klemens Böhm. Efficient evaluation of nearest-neighbor queries in content-addressable networks. In *From Integrated Publication and Information Systems to Virtual Information and Knowledge Environments*, pages 31–40, 2005.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The pyramid-technique: towards breaking the curse of dimensionality. *SIGMOD Rec.*, 27(2):142–153, 1998.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[BCG05] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.

[BEF+09] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, abs/0905.4627v2, 2009.

[Ben90] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Symposium on Computational Geometry*, pages 187–197, 1990.

[BGRS99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, pages 217–235, 1999.

[BKS01] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[BKS04] Farnoush Banaei-Kashani and Cyrus Shahabi. Swam: a family of access methods for similarity-search in peer-to-peer data networks. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 304–313, New York, NY, USA, 2004. ACM.

[BKST78] Jon Louis Bentley, H. T. Kung, Mario Schkolnick, and Clark D. Thompson. On the average number of maxima in a set of vectors and applications. *J. ACM*, 25(4):536–543, 1978.

[BM96] Timothy A. H. Bell and Alistair Moffat. The design of a high performance information filtering system. In Hans-Peter Frei, Donna Harman, Peter Schäuble, and Ross Wilkinson, editors, *SIGIR*, pages 12–20. ACM, 1996.

[BMS+06] Holger Bast, Debapriyo Majumdar, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[BOPY07] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, pages 156–165, 2007.

[BSW04] Shivnath Babu, Utkarsh Srivastava, and Jennifer Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Trans. Database Syst.*, 29(3):545–580, 2004.

[Cal96] James P. Callan. Document filtering with inference networks. In *SIGIR*, pages 262–269, 1996.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In Chen et al. [CNB00], pages 379–390.

[CG07] Graham Cormode and Minos N. Garofalakis. Sketching probabilistic data streams. In *SIGMOD Conference*, pages 281–292, 2007.

[CKT08] Graham Cormode, Flip Korn, and Srikanta Tirthapura. Time-decaying aggregates in out-of-order streams. In *PODS*, pages 89–98, 2008.

[CLM+07] Adina Crainiceanu, Prakash Linga, Ashwin Machanavajjhala, Johannes Gehrke, and Jayavel Shanmugasundaram. P-ring: an efficient and robust p2p range index structure. In *SIGMOD Conference*, pages 223–234, 2007.

[CM03] Graham Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *PODS*, pages 296–306, 2003.

[CNB00] Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. ACM, 2000.

[CRB+03] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making gnutella-like p2p systems scalable. In Feldmann et al. [FZCW03], pages 407–418.

[CW04] Pei Cao and Zhe Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.

[DF03] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities (extended abstract). In *ESA*, pages 605–617, 2003.

[DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.

[DGKS07] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Nikos Sarkas. Ad-hoc top-k query answering for data streams. In *VLDB*, pages 183–194, 2007.

[DGKT06] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *VLDB*, pages 451–462. ACM, 2006.

[DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD Conference*, pages 40–51, 2003.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[DVKV07] Christos Doulkeridis, Akrivi Vlachou, Yannis Kotidis, and Michalis Vazirgiannis. Peer-to-peer similarity search in metric spaces. In *VLDB*, pages 986–997, 2007.

[Fag02] Ronald Fagin. Combining fuzzy information: an overview. *SIGMOD Record*, 31(2):109–118, 2002.

[FGZ05] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. A content-addressable network for similarity search in metric spaces. In *DBISP2P*, pages 98–110, 2005.

[FLN03] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[FM85] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[FSGM+98] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *VLDB*, pages 299–310, 1998.

[FZCW03] Anja Feldmann, Martina Zitterbart, Jon Crowcroft, and David Wetherall, editors. *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany*. ACM, 2003.

[GBK00] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *VLDB*, pages 419–428, 2000.

[Gib01] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550, 2001.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[GKMS01] Anna C. Gilbert, Yannis Kotidis, S. Muthu Muthukrishnan, and Martin Strauss. Quicksand:quick summary and analysis of network data. Technical report, December 2001.

[GMP97] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. In *VLDB*, pages 466–475, 1997.

[Gnua] Gnutella. http: //www.gnutelliums.com/.

[Gnub] The gnutella protocol specification v0.6. http://rfcgnutella.sourceforge.net.

[GÖ03] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[GT02] Phillip B. Gibbons and Srikanta Tirthapura. Distributed streams algorithms for sliding windows. In *SPAA*, pages 63–72, 2002.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[HMA09a] Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *EDBT*, pages 744–755, 2009.

[HMA09b] Parisa Haghani, Sebastian Michel, and Karl Aberer. Evaluating top-k queries over incomplete data streams. In *CIKM*, pages 877–886, 2009.

[HMA10] Parisa Haghani, Sebastian Michel, and Karl Aberer. The gist of everything new: Personalized top-k processing over web 2.0 streams. In *to appear in CIKM*, 2010.

[HMCMA08] Parisa Haghani, Sebastian Michel, Philippe Cudré-Mauroux, and Karl Aberer. Lsh at large - distributed knn search in high dimensions. In *WebDB*, 2008.

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

[JMB05] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, 2005.

[JOT+05] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang 0003. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.

[JOV+06] H. V. Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 34, Washington, DC, USA, 2006. IEEE Computer Society.

[JY+08] Cheqing Jin, Ke Yi, Lei Chen 0002, Jeffrey Xu Yu, and Xuemin Lin. Sliding-window top-k queries on uncertain streams. *PVLDB*, 1(1):301–312, 2008.

[KM00] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In Chen et al. [CNB00], pages 201–212.

[KOT04] Nick Koudas, Beng Chin Ooi, Kian-Lee Tan, and Rui Zhang 0003. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB*, pages 804–815, 2004.

[KPSV09] Ravi Kumar, Kunal Punera, Torsten Suel, and Sergei Vassilvitskii. Top- aggregation using intersections of ranked inputs. In Ricardo A. Baeza-Yates, Paolo Boldi, Berthier A. Ribeiro-Neto, and Berkant Barla Cambazoglu, editors, *WSDM*, pages 222–231. ACM, 2009.

[LCKB06] Feifei Li, Ching Chang, George Kollios, and Azer Bestavros. Characterizing and exploiting reference locality in data stream applications. In *ICDE*, page 81, 2006.

[LJW+07] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[MAA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In Thomas Eiter and Leonid Libkin, editors, *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[MAAZ07] Ahmed Metwally, Divyakant Agrawal, Amr El Abbadi, and Qi Zheng. On hit inflation techniques and detection in streams of web advertising networks. In *ICDCS*, page 52. IEEE Computer Society, 2007.

[MBP06] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006.

[MP07] Kyriakos Mouratidis and Dimitris Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6):789–803, 2007.

[MP09] Kyriakos Mouratidis and HweeHwa Pang. An incremental threshold method for continuous text search queries. In *ICDE*, pages 1187–1190, 2009.

[MSL+09] Sebastian Michel, Ali Salehi, Liqian Luo, Nicholas Dawes, Karl Aberer, Guillermo Barrenetxea, Mathias Bavay, Aman Kansal, K. Ashwin Kumar, Suman Nath, Marc Parlange, Stewart Tansley, Catharine van Ingen, Feng Zhao, and Yongluan Zhou. Environmental monitoring 2.0. In *ICDE*, pages 1507–1510, 2009.

[MTW05a] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB*, pages 637–648, 2005.

[MTW05b] Sebastian Michel, Peter Triantafillou, and Gerhard Weikum. Minerva$_{\text{infinity}}$: A scalable efficient peer-to-peer search engine. In *Middleware*, pages 60–81, 2005.

[Mut05] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[NR99] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *ICDE*, pages 22–29. IEEE Computer Society, 1999.

[ORC+98] Michael Ortega, Yong Rui, Kaushik Chakrabarti, Kriengkrai Porkaew, Sharad Mehrotra, and Thomas S. Huang. Supporting ranked boolean similarity queries in mars. *IEEE Trans. Knowl. Data Eng.*, 10(6):905–925, 1998.

[Pan06] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, pages 1186–1195, 2006.

[PNT06] Theoni Pitoura, Nikos Ntarmos, and Peter Triantafillou. Replication, load balancing and efficient range query processing in dhts. In *EDBT*, pages 131–148, 2006.

[PT07] Theoni Pitoura and Peter Triantafillou. Load distribution fairness in p2p data management systems. In *ICDE*, pages 396–405, 2007.

[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 *DEBS*, pages 127–138, 2008.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.

[Roc98] R. Tyrrell Rockafellar. *Network Flows and Monotropic Optimization*. John Wiley and Sons, 1998.

[SAA00] Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[SEAA04] Ozgur D. Sahin, Fatih Emekçi, Divyakant Agrawal, and Amr El Abbadi. Content-based similarity search over peer-to-peer systems. In *DBISP2P*, pages 61–78, 2004.

[Sel08] Margo I. Seltzer. Beyond relational databases. *Commun. ACM*, 51(7):52–58, 2008.

[SFT03] Amit Singh, Hakan Ferhatosmanoglu, and Ali Saman Tosun. High dimensional reverse nearest neighbor queries. In *CIKM*, pages 91–98. ACM, 2003.

[SH98] Mark Sullivan and Andrew Heybey. Tribeca: a system for managing large databases of network traffic. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[SMK+01] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[SW04] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 324–335. Morgan Kaufmann, 2004.

[Swi] Swiss experiment: http://www.swiss-experiment.ch/.

[TKD09] Christos Tryfonopoulos, Manolis Koubarakis, and Yannis Drougas. Information filtering and query indexing for an information retrieval model. *ACM Trans. Inf. Syst.*, 27(2), 2009.

[TWS04] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB*, pages 648–659, 2004.

[TXB06] Srikanta Tirthapura, Bojian Xu, and Costas Busch. Sketching asynchronous streams over a sliding window. In *PODC*, pages 82–91, 2006.

[TXD03] Chunqiang Tang, Zhichen Xu, and Sandhya Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In Feldmann et al. [FZCW03], pages 175–186.

[TYSK09] Yufei Tao, Ke Yi, Cheng Sheng, and Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *SIGMOD Conference*, pages 563–576. ACM, 2009.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[XYC05] Junyi Xie, Jun Yang, and Yuguo Chen. On joining and caching stochastic streams. In *SIGMOD Conference*, pages 359–370, 2005.

[YGM94a] Tak W. Yan and Hector Garcia-Molina. Index structures for information filtering under the vector space model. In *ICDE*, pages 337–347, 1994.

[YGM94b] 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, 1994.

[YOTJ01] Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[YYY+03] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *ICDE*, pages 189–200. IEEE Computer Society, 2003.

[ZKW04] Chi Zhang, Arvind Krishnamurthy, and Randolph Y. Wang. Skipindex: Towards a scalable peer-to-peer index service for high dimensional data. Technical report, Department of Computer Science, Princeton University, May 2004.

[ZS02] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *VLDB*, pages 358–369. Morgan Kaufmann, 2002.

Curriculum Vitæ

Personal Data

Name: Parisa Haghani

Date of Birth: 31 May 1981

Place of Birth: Shiraz, Iran

Nationality: Iranian

Languages: English (fluent), French (intermediate), Persian (native)

Education

PhD in Computer Science (2010)

Ecole Polytechnique Federale de Lausanne (EPFL), Switzerland.

PhD thesis: Efficient Processing of Ranking Queries in Novel Applications.

MSc in Computer Engineering (2006)

Sharif University of Technology (SUT), Iran.

MSc thesis: Developing a new Evidence Theory based on the diffusion operator

BSc in Computer Engineering (2004)

Isfahan University of Technology, Iran.

BSc Project: Security in Ad-hoc Networks

Honors and Awards

Graduated with rank 1 among M. Sc. Students in Artificial Intelligence, Sharif University of Technology, Sharif, Iran, 2006.

Ranked 6th among approximately 2000 participants in the National Graduate Examination for Computer Engineering, Iran, 2004.

Graduated with rank 1 among 60 B. Sc. Students in Computer Engineering, Isfahan University of Technology, Isfahan, Iran, 2004.

Member of Sahand Soccer Simulation Team which took part in International Robocup Competitions, Italy, 2003.

Ranked 290 among approximately 350,000 in the National Entrance Examination for Iranian Universities, Iran, 2000.

Achieving a score above 10500 in the National Entrance Examination and becoming a member of National Organization for Developing Exceptional Talents, Isfahan University of Technology, Isfahan, Iran, 2000.

Publications

1. Parisa Haghani, Sebastian Michel, Karl Aberer: Evaluating top-k queries over incomplete data streams. 18th ACM Conference on Information and Knowledge Management (CIKM2010), Toronto, Canada, October 26-30, 2010.

2. Parisa Haghani, Sebastian Michel, Karl Aberer: Evaluating top-k queries over incomplete data streams. 18th ACM Conference on Information and Knowledge Management (CIKM2009), HongKong, China, November 2-6, 2009.

3. Philippe Cudre-Mauroux , Parisa Haghani, Michael Jost, Karl Aberer, Hermann de Meer: idMesh: Graph-Based Disambiguation of Linked Data. 18th International World Wide Web Conference (WWW2009), Madrid, Spain, April 20-24, 2009.

4. Parisa Haghani, Sebastian Michel, Karl Aberer: Distributed Similarity Search in High Dimensions Using Locality Sensitive Hashing. 12th International Conference on Extending Database Technology (EDBT 2009), Saint-Petersburg, Russia, March 23-26, 2009.

5. Parisa Haghani, Sebastian Michel, Philippe Cudre-Mauroux, Karl Aberer:. LSH At Large - Distributed KNN Search in High Dimensions. 11th International Workshop on Web and Databases (WebDB 2008), Vancouver, Canada, June 13, 2008.

6. Parisa Haghani, Panos Papadimitratos, Marcin Poturalski, Karl Aberer, Jean-Pierre Hubaux: Efficient and Robust Secure Aggregation for Sensor Networks. The 3rd workshop on Secure Network Protocols (NPSec 2007) Beijin, China, October 2007.

7. Philippe Cudre-Mauroux, Suchit Agarwal,Adriana Budura, Parisa Haghani, Karl Aberer: Self-Organizing Schema Mappings in the GridVine Peer Data Management System. The 33rd International Conference on Very Large Data Bases (VLDB 2007).

Work and Teaching Experience

Research Assistant (EPFL) 2006-present

Distributed Information Systems Lab.

Teaching Assistant (EPFL) Fall 2008

Distributed Information Systems

Teaching Assistant (SUT) 2004-2006

Discrete Structures

Programming Course (C++)

Theory of Languages and Automata