Continuous Similarity Search for Evolving Queries

by

Xiaoning Xu

B.Sc., Simon Fraser University, 2012

B.Eng., Zhejiang University, 2012

a Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

APPROVAL

Name:	Xiaoning Xu
Degree:	Master of Science
Title of Thesis:	Continuous Similarity Search for Evolving Queries

Date Approved: April 28th, 2014

Abstract

Despite the vast researches on database similarity search/join, little work has been embarked on the similarity search problem when the query is evolving. How can we select advertisements for users who recently registered at a Q&A website to capture the evolving trend of user interest? How can we continuously recommend songs by user humming? Motivated by the above application scenarios, we study a novel problem of continuous similarity search for evolving queries. Given a set of objects, each being a set or multi-set of items, and a data stream where each element is from the same lexicon as the objects, we want to continuously maintain the top-k most similar objects using the last n items in the stream as an evolving query. We develop a filtering-based method and a MinHash-based method. Our experimental results on real data sets and synthetic data sets show that our methods are effective and efficient.

To my parents.

"You will recognize your own path when you come upon it, because you will suddenly have all the energy and imagination you will ever need."

— Jerry Gillies, *(1940-)*

Acknowledgments

I would like to express my deepest gratitude to my senior supervisor, Dr. Jian Pei, for his great patience, warm encouragement and continuous support throughout my Master's studies. His wisdom and passion in research has influenced and inspired me a lot, which gave me confidence and interest in accomplishing this thesis.

I would like to thank Dr. Ke Wang for being my supervisor and giving me helpful suggestions on my thesis. I also thank Dr. Anoop Sarkar and Dr. Jiangchuan Liu for serving in my examining committee.

I am also very grateful to my lab mates for their kind help. A special thank goes to Chuancong Gao, Juhua Hu, Lei Duan, Guanting Tang, Xiangbo Mao, Xiao Meng, Ye Wang, Yu Tao, Jiaxing Liang, Li Xiong, Lin Liu, Beier Lu, Yi Zhuang, and Yu Yang.

Moreover, my sincerest gratitude goes to my parents for their endless love and support through all these years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we first discuss several interesting applications that motivate the problem of continuous similarity search for evolving queries, which will be studied in this thesis. Then, we will summarize the major contributions and describe the structure of the thesis.

1.1 Motivations

Let us consider a cold-start recommendation problem at a Q&A website. Suppose you want to run 3 advertisements on the Q&A website for a user who does not have a profile yet, what advertisements should you display? In addition to many factors, such as click-through rates of advertisements and bidding price information, a natural and important idea is to consider the advertisements that are most related to the questions being recently asked at the website by all users. For example, you may want to retrieve the top-k advertisements that are most similar to the last n questions asked, where the similarity measure captures the relevance between advertisements and questions. Such advertisements can be used as the candidates for further selection.

The above scenario is just one of the many applications that motivate the problem to be studied in this thesis. Given a set of static data objects (e.g., ads in the above example), and an evolving data stream (e.g., the questions asked in the above example), a sliding window on the data stream (e.g., the last n questions in the above example) presents an evolving query. The problem of *continuous similarity search for evolving queries* is to continuously conduct top-k similarity search on the set of static objects using the evolving queries.

Table 1.1: Motivation example: keywords on advertisements

ad_1	{sneaker, outdoor, low-top, Nike, men's}
ad_2	{insurance, donate, treatment, red cross, cord blood}
ad_3	{TransLink, uPass, Compass Card, skytrain}
ad_4	{nikon, lens, megapixels, slr camera}
ad_5	{Apple, iPhone 5s, A7 chip, fingerprint sensor}
ad_6	{Nexus, tablet, Google, Android, resolution}

Example 1.1 (Motivation Example)**.** *We are given a set of keywords or phrases for each bidding advertisement as shown in Table 1.1. At the current moment, assume that the keywords and phrases extracted from the last* 10 *questions asked by users in the Q&A website are {insurance, red cross, A7 chip, fingerprint sensor, Apple, iOS 7, Android, uPass}. Without any further background information, we may assume that a new user may also be interested in some of the topics related to the recent questions asked by all users. Thus, to place advertisements for new users, we simply find the advertisements whose keywords or phrases are, to some extent, similar to those extracted from the recent questions. Suppose we always display the top-*2 *most similar advertisements due to limited space on a webpage and quality user experience, the advertisements displayed for newly-registered users would be ad_5 and ad_2 based on a predefined similarity measurement, for example, overlap similarity or Jaccard similarity. After a short period of time, say a second, if the keywords and phrases of the most recent* 10 *questions have been updated to {fingerprint sensor, Apple, iOS 7, Android, uPass, sneaker, Compass Card, TransLink}. The advertisements recommended for the new users now would be changed to ad_5 and ad_3.*

This problem of continuous similarity search for evolving queries has many important applications. As another example, in a computer war game, a virtual player has a set of weapons and tools, which is relatively stable. The player goes through a virtual reality space, where the objects in the continuously updated surrounding environment, such as different types of enemies, scoring opportunities, and obstacles, present a stream of evolving queries. The virtual players has to select proper weapons and tools that match the current surrounding environment best. Again, before any gaming strategies can be used, an essential task is to continuously maintain the top-k best weapons and tools with respect to the evolving queries.

Another interesting application with a similar nature is continuous music recommendation by user humming. A popular service in Apple app store called *Shazam* can identify a song by taking a short sample of the music. It stores the fingerprints of a comprehensive catalog of music in a database. A song is returned if there is a match in the database regarding the fingerprint of the sample of the song uploaded by the user. However, *Shazam* cannot suggest songs continuously if we play small fractions of several songs in a consecutive way. Thus, we would like to find out a solution to suggest songs continuously using the most recent part of a music stream that consists of small fractions of multiple songs as the query. This application is in demand quite often in daily life. For example, suppose we want to request multiple songs in a KTV song request system and cannot remember the name of each song clearly. Instead of checking the exact song names and searching them in the song request system, it would be much more convenient if the system can suggest the songs continuously when we sing a small part of each song consecutively.

Our problem can be considered as the dynamic version of the top-k set similarity search problem. More generally, a traditional similarity search problem involves a collection of objects, a similarity function, and a user defined threshold. The search is to find all the objects in the collection whose similarity scores regarding the query are higher than the predefined threshold. Similarity search has many applications, such as information retrieval [40, 20, 18], near-duplicate web page detection [26], record linkage [43], data compression [20], data integration [11], image and video search and recommendation [17, 19, 39, 41], statistics and data analysis [16, 29], machine learning [12], and data mining [25, 3]. Besides answering threshold-based queries, top-k queries are also of great value since given a threshold, the size of the result may be unpredictable and, in many real applications, we are only interested in a small number of most similar objects. Moreover, as to be reviewed in Section 2.2, the problem of similarity search on data streams has been extensively explored, especially for nearest neighbour search. However, to the best of our knowledge, the problem of continuous similarity search for evolving queries has not been systematically investigated.

1.2 Contributions

In this thesis, we tackle the problem of continuous similarity search for evolving queries. Since in many applications, an object can be represented as a multi-set, such as using a keyword vector to represent a document, we consider static objects as multi-sets, and use weighted Jaccard similarity as the measure. The major challenge is how to speed up the similarity computation and avoid checking evolving queries with every static object exactly at every time point. We make the following contributions.

- We formulate the problem of *continuous similarity search for evolving queries* that continuously finds the top-k objects in a collection of sets that are most similar to an evolving query.
- We develop upper bounds for incremental maintenance of similarity scores. Those bounds can be computed in constant time.
- We propose algorithms based on two general frameworks. The first one is *pruning and verification*. The other one is *hashing*. The pruning-based method reduces the cost of computing the exact similarity scores using pruning strategies. The MinHash-based method estimates Jaccard similarity scores based on MinHash technique [6] and uses indexing structures for efficient updates.
- We report an empirical evaluation on both synthetic and real-world data sets, which validates the efficiency and effectiveness of our proposed methods.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. In Chapter 2, we review the related work. We then formulate the problem of *continuous similarity search for evolving queries* in Chapter 3. In Chapter 4, we propose a pruning-based method. In Chapter 5, we present a MinHash-based method. We report our experimental results in Chapter 6, and conclude the thesis in Chapter 7.

Chapter 2

Related Work

Our problem of *continuous similarity search for evolving queries* is mainly related to the existing work on similarity search and continuous top-k queries, which are reviewed in Section 2.1 and Section 2.2, respectively.

2.1 Similarity Join and Search

The static version of similarity search has been studied extensively. The state-of-the-art methods can be categorized into two major groups, namely, a filtering-based framework and a hashing-based framework.

2.1.1 Filtering-based Methods

The general idea of filtering-based approaches is to develop efficient and effective filters to prune objects whose similarity, regarding the query, cannot be larger than a threshold. Consequently, the exact similarity scores, which are supposed to be more expensive to compute, are calculated for only a small number of surviving objects from filtering.

Similarity join and search over sets. Chaudhuri *et al.* [9] developed the prefix-filtering principle for similarity joins. The intuitive idea is that if two objects are similar, they tend to have overlap in a portion of the records. The main strategy is to prune object pairs whose prefix sets, according to a global ordering, do not have any overlap and thus the similarity between the original objects is guaranteed to be smaller than the similarity threshold. The filtering phase is followed by a verification phase where similarity between the surviving objects and the query is computed. Bayardo *et al.* [3] proposed the *All-Pairs* algorithm that further improves the prefix-filtering approach using various refinements, such as tight transformation regarding threshold constraints between overlap similarity and other similarity measures, an index reduction principle on prefix size, and a length filtering scheme. Xiao *et al.* [46] devised a positional filtering approach, ppjoin+, which makes use of the token ordering information. By applying this method on the prefix and the suffix of an object, the number of candidate pairs can be reduced significantly. Recently, Wang *et al.* [42] proposed an adaptive framework to select prefix for each object based on the observation that the computational cost for each object may not be the lowest when the prefix length is fixed.

Similarity join and search over strings. Most of the current methods for string similarity join and search are focused on developing filtering techniques that operate on q-gram representation of strings that transforms a string to a set and use edit distance as the similarity constraint. Gravano *et al.* [22] first introduced the idea of q-gram representation, which is a set of substrings obtained by sliding a window of length q over the original string. Three types of filtering strategies are used, namely, count filtering, position filtering, and length filtering. Instead of matching q-grams using the traditional filtering methods, a new perspective by studying mismatching q-grams was presented in [44]. Besides that, they also proposed two new filtering methods: location-based mismatch filtering and content-based mismatch filtering, to detect the minimum edit errors and the cluster edit errors, respectively. There are also some other filtering approaches without using q-grams. For example, Li *et al.* [32] proposed a segment filter that first partitions the strings and builds indices over the segments. These segments are used as filters with the intuition that dissimilar string pairs share none or limited number of segments. More interestingly, Chen *et al.* [10] developed pruning strategies that use triangle inequality for large time series databases. The metric distance function used is a combination of L1-norm and edit distance, which is called ERP ("Edit distance with Real Penalty"). The idea of this method can be used for edit distance since it satisfies triangle inequality.

Top-k similarity join and search on sets and strings. Besides methods for threshold-based queries as discussed above, some work has been done on top-k similarity join and search for both sets and strings. The main technical challenge of answering top-k queries is that the k-th largest similarity score is unknown. Xiao [45] studied the problem of top-k set similarity join based on the prefix-filtering framework and the incremental execution of the *All-Pairs* algorithm. Some optimization techniques, such as the monotonicity of the k-th largest similarity seen so far, are applied. As for top-k string similarity search, Deng [14] designed a progressive framework to compute edit distance efficiently and extended it to support top-k similarity search.

Nearest neighbour search in d-dimensional Euclidean space. Some typical indexing structures for nearest neighbour search are k-d tree [4] and R tree [23]. A k-d tree is built by iteratively partitioning the search space into two regions, each containing half of the points of the parent region. The idea of R tree is to bound objects using minimum bounding rectangles and thus group close objects together. Both tree structures can be used to prune objects that are far away from the query point. In our problem, objects are modeled as sets instead of points in Euclidean space. Since most of the previous indexing structures are based on space partitioning, it is not feasible to apply to our problem.

2.1.2 Hashing-based Methods

The second category is hashing-based methods. The general idea is to develop hash functions that have good locality preservation properties – similar objects are likely hashed to the same bucket, which is formally introduced in [28] for approximate nearest neighbour search in d-dimensional Euclidean space. The basic principle is to hash the points using multiple hash functions such that closer points have higher probability of collision than points that are far away. Gionis *et al.* [21] further improved the algorithms and achieved better query time guarantees. Later, an improved algorithm that almost achieves the space and time lower bounds is presented in [2]. The MinHash technique [6] is used to approximate the resemblance and the containment of sets. This technique is used to estimate the rarity and similarity between two windowed data streams in [13]. Moreover, Charikar [8] proposed SimHash to hash similar data to similar values. An estimation for vector-based cosine similarity using a random projection method is also discussed.

In this thesis, we explore both filtering-based and hashing-based methods, which have not been addressed in the existing literature for evolving queries.

2.2 Continuous Queries over a Data Stream

In this section, we briefly review various data stream models. Since our problem shares many common characteristics with continuous query answering over a data stream, we then review related problems including but not limited to similarity search.

2.2.1 Data Stream Models

A data stream is a sequence of data elements $a_1, a_2, \ldots$ that arrives item by item, which describes an underlying signal. The signal is a one-dimensional function $A : [1...N] \mapsto R$, where the domain consists of integers and the range is the set of real numbers. As mentioned in [37], there are mainly three models that differ on how items describe the signal, namely, time series model, turnstile model, and cash register model. We summarize the models briefly as the following.

- **Time Series Model** This is the simplest model in which each a_i is equal to $A[i]$ and items show up in the increasing order of i. We can think of the daily closing price of a stock from the listing date to the present as a data stream. Each a_i corresponds to the closing price of the stock at day i.

- **Turnstile Model** In this most general model, $a_i = (j, U_i)$ which is an update to $A[j]$ where U_i can be any real number. Suppose A_i is the signal after processing the i^{th} item a_i in the stream, the update can be written as $A_i[j] = A_{i-1}[j] + U_i$. Let us consider a scenario where N investors buy and sell shares of a certain stock. Since investors can buy and sell stock without particular order, a data stream, i.e., a sequence of transactions, $a_1, a_2, \ldots,$ can be generated. The i^{th} transaction $a_i = (j, U_i)$ indicates that investor j wants to buy $|U_i|$ shares if $U_i > 0$ and sell $|U_i|$ shares otherwise. Then, the total shares held by investor j is updated by $A_i[j] = A_{i-1}[j] + U_i$.

- **Cash Register Model** It can be considered as a special case of the Turnstile model where $U_i \geq 0$. To be more specific, each $a_i = (j, U_i)$ is an increment instead of a general update to $A[j]$. Following the previous example, if investors can only buy shares of stocks, the transaction sequence fits the cash register model.

Many streaming algorithms are designed for the entire data stream. However, a wide range of real applications, such as web log mining and stock market prediction, do not consider outdated elements important. Thus, the sliding window model is of great importance. In this model, we only consider the most recent part of the data stream. Typically, there are two types of sliding windows with equal importance. The count-based sliding window is of fixed size and contains the last n items in the data stream. The time-based sliding window allows bursts at a single time unit since it contains the items that arrived in the last n time units. In this thesis, we focus on the count-based sliding window.

2.2.2 Answering Continuous Queries

Different evolving models are used in previous studies that investigated continuous queries over a data stream. For example, Kontaki *et al.* [30] studied similarity range queries in streaming time sequences using Euclidean distance, where both the query and data objects are evolving. An indexing method that is based on incremental computation method for Discrete Fourier Transform is used for achieving high candidates ratio. Lian *et al.* [33] tackled the similarity search problem over multiple stream time series, given a static time series as a query. An approximation algorithm is developed using a weighted locality sensitive hashing technique.

Motivated by a wide range of applications such as network intrusion detection, much work [5, 31, 35, 36] has been embarked on monitoring nearest neighbour (NN) queries continuously over a data stream. The basic idea is to utilize indexing structures for reducing memory consumption and supporting efficient updates. Mouratidis *et al.* [35] proposed two approaches for continuous monitoring of NN queries over sliding window streams. The first approach extends the conceptual partitioning method to the sliding window model. Skyline maintenance techniques and pre-computation of future changes in nearest neighbours are used in the second approach. Koudas *et al.* [31] developed an approximation algorithm that utilizes an indexing scheme, DISC, and has guaranteed error and performance bound.

The existing work on continuously monitoring nearest neighbours for mobile query object is different from the problem studied here. In those previous studies, the mobile object is assumed to move in a trajectory, potentially predictable to some extent. In this thesis, the stream presenting an evolving query is not assumed a moving object. Instead, we simply use the current sliding window as the current query. The existing methods on continuous nearest neighbour monitoring for mobile objects cannot solve our problem.

Besides continuous queries on similarity search problems, some interesting work is done for graph streams and general functions defined over data streams. Pan and Zhu [38] developed a two-level candidate checking scheme for continuously querying the top-k correlated graphs in a data stream scenario where static queries are posed on evolving graph streams. Mouratidis *et al.* [34] proposed two approaches for continuously answering top-k queries where the query is a static preference function over a fixed-sized sliding window. One approach is to compute new answers whenever a current top-k point expired and the other approach is to precompute future changes partially.

Our proposed methods aim at providing algorithmic frameworks on how to deal with similarity search over evolving queries. In our problem, we can take advantage of the fact that consecutive queries in a stream are very similar. This results in some important design in our algorithms, which distinguishes our methods from the previous ones. For example, we derive an upper bound on similarity scores after a few updates based on how the queries evolve. The pruning-based algorithm uses this bound to prune unpromising candidates. In the hashing-based algorithm, we maintain fixed-length signatures for all transactions and a query. Since consecutive queries share a large portion of items, the change in the signature of the query is small. With the help of inverted index, we can further speed up the updates of similarity scores.

Chapter 3

Problem Definition

Given a collection of objects, a query object and a similarity measure, the similarity search problem is to find all objects whose similarity scores, with respect to the query, are higher than a threshold. We consider a variant of the similarity search problem. One variation is that, instead of answering a query according to a threshold, we return k most similar objects as the query result.

Figure 3.1: An Example on Count-based Sliding Window

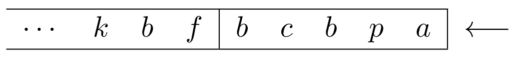
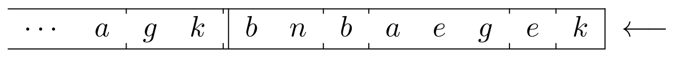


Figure 3.2: An Example on Time-based Sliding Window


$\cdots$ a g k b n b a e g e k $\longleftarrow$

We then put this problem in the data stream context and want to answer this query continuously based on the most recent part of the data stream, *i.e.*, a sliding window. There are generally two types of sliding windows, namely, count-based window and time-based window. The count-based sliding window contains the last n items in the data stream while the time-based sliding window contains the items that arrived in the last n time units. For example, suppose $n = 5$ and the lexicon of the stream consists of all the letters in the English alphabet, we show the count-based window and time-based window at time t in Figure 3.1 and 3.2, respectively. The count-based sliding window consists of 5 items, namely, $\{b, c, b, p, a\}$. In Figure 3.2, we use vertical dash line to separate items came at different time units. The time-based sliding window is made up of items came during the

Table 3.1: A collection of sets $\mathcal{R}$

r_1	e_2	e_1	e_3	e_6	e_9	
r_2	e_1	e_5	e_4			
r_3	e_4	e_5				
r_4	e_2	e_9	e_3	e_4		
r_5	e_7	e_8	e_9	e_{10}	e_3	e_1
r_6	e_1	e_5	e_6	e_3	e_8	

Figure 3.3: Query stream $\mathcal{S}$ and query q

$$\cdots \quad \boxed{e_2 \quad e_9 \quad e_3 \quad e_1 \quad e_4} \longleftarrow$$

Figure 3.4: Query stream $\mathcal{S}$ after a new item arrives, presenting query q'

$$\cdots \quad e_2 \quad \boxed{e_9 \quad e_3 \quad e_1 \quad e_4 \quad e_6} \longleftarrow$$

most recent 5 time units, which is $\{b, n, b, a, e, g, e, k\}$. Our definitions and methods are illustrated in the count-based model. We now define the problem formally.

Definition 3.1 (Continuous Top-k Queries)**.** *Let us consider an alphabet of items Σ, a collection of objects $\mathcal{R}$ where each object is a set over Σ, a data stream $\mathcal{S}$ with elements $e \in \Sigma$ kept coming, and an integer n as the size of a query sliding window on $\mathcal{S}$. A query q is the last n elements in $\mathcal{S}$. The top-k continuous similarity search for evolving query q is to continuously find the k objects in $\mathcal{R}$ that are most similar to q. We answer the query continuously whenever a new element arrives in $\mathcal{S}$.*

Example 3.1 (Continuous Top-k Queries)**.** *A collection of sets $\mathcal{R}$ and a data stream $\mathcal{S}$ are shown in Table 3.1 and Figure 3.3, respectively. The alphabet $\Sigma = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$. Suppose $k = 2$ and the size of the sliding window is 5, that is, we use the last 5 entries in stream $\mathcal{S}$ as the evolving query, and find the top-2 most similar sets in $\mathcal{R}$.*

We can compute the Jaccard similarity score between an object in $\mathcal{R}$ and the current query $q = \{e_2, e_9, e_3, e_1, e_4\}$. The scores are shown in the first row of Table 3.2. Objects r_4 and r_1 are the top-2 similar objects with respect to query q.

Suppose a next item e_6 arrives at the data stream as shown in Figure 3.4, we have an updated query $q' = \{e_9, e_3, e_1, e_4, e_6\}$. The updated Jaccard similarity scores are shown in the last row of Table 3.2. The rankings of the objects in $\mathcal{R}$ ordered by similarity scores change

Table 3.2: Jaccard similarity scores

	r_1	r_2	r_3	r_4	r_5	r_6
q	**0.67**	0.33	0.17	**0.80**	0.38	0.25
q'	**0.67**	0.33	0.17	**0.50**	0.38	0.43

slightly. For example, the rank of r_4 *changes from* 1 *to* 2*. As for the query result,* r_1 *and* r_4 *are the top-*2 *similar objects with respect to query* q'*.*

In this thesis, we consider each object being a set. The queries can be sets or multi-sets. Since a set is a special case of a multi-set when elements in the collection are all distinct, we use weighted Jaccard similarity to measure the similarity between objects and queries.

Specifically, let X be a multi-set that consists of items in alphabet $\Sigma = \{e_1, e_2, ..., e_{|\Sigma|}\}$. We map X to a $|\Sigma|$-dimensional vector $\vec{X}$ such that the value of the i-th component is the absolute frequency of item e_i in X. The vector $\vec{X}$ is called the *vector representation* of multi-set X.

Let X and Y be two non-empty multi-sets. X and Y both consist of elements in alphabet Σ. Let $\vec{X} = \{x_1, x_2, ..., x_{|\Sigma|}\}$ and $\vec{Y} = \{y_1, y_2, ..., y_{|\Sigma|}\}$ be the vector representations of X and Y, respectively. The weighted Jaccard similarity is defined as

$$sim_{Jac}(X, Y) = sim_{Jac}(\vec{X}, \vec{Y}) = \frac{\sum_{i=1}^{|\Sigma|} \min(x_i, y_i)}{\sum_{i=1}^{|\Sigma|} \max(x_i, y_i)}$$

Example 3.2 (Computing Similarity Scores)**.** *Given an alphabet* $\Sigma = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$*, we have* $X = \{e_3, e_2, e_1, e_5, e_6\}$ *and* $Y = \{e_1, e_2, e_1, e_3, e_5, e_4\}$*. We can first transform* X *and* Y *to vectors, that is,* $\vec{X} = \{1, 1, 1, 0, 1, 1, 0\}$ *and* $\vec{Y} = \{2, 1, 1, 1, 1, 0, 0\}$*. Since* $\sum_{i=1}^{7} \max(x_i, y_i) = 7$ *and* $\sum_{i=1}^{7} \min(x_i, y_i) = 4$*, the weighted Jaccard similarity score between* X *and* Y *is* $sim_{Jac}(X, Y) = \frac{4}{7} = 0.57$*.*

Chapter 4

A Pruning-based Method

Computing the exact similarity score for every object against the updated query at every time point is costly. We derive an upper bound of Jaccard similarity scores in Section 4.1, and then present a pruning-based method that utilizes the upper bounds in Section 4.2. In Section 4.3, we claim that the proposed method can be used as a framework for many other similarity measures including Cosine similarity and Edit similarity. We review the definitions of these similarity measures and then derive the upper bounds that can be computed in constant time.

4.1 A Progressive Upper Bound

We define the length of a multiset as the number of elements, including duplicates, in this multiset. Let us denote the lengths of an object and the query as $|X|$ and $|Y|$, respectively. Based on the definition of count-based sliding windows, given a query $q = \{e_1, e_2, ..., e_{|q|}\}$, the updated query after u time instants is $q' = \{e_{u+1}, e_{u+2}, ..., e_{|q|}, e_{new_1}, e_{new_2}, ..., e_{new_u}\}$, where u is a positive integer and $u < |q|$.

To compute the upper bound for Jaccard similarity score after u updates, we can use the following property.

Property 4.1 (A Progressive Upper Bound for Jaccard Similarity)**.** *Let X, Y be two multisets and Y'_u be the multiset with u updates on Y. Given $|X|$, $|Y|$, and the weighted Jaccard similarity score $sim_{Jac}(X, Y)$ between X and Y, without the knowledge of the updated elements in Y'_u, we have an upper bound for $sim_{Jac}(X, Y'_u)$.*

$$sim_{Jac}(X, Y'_u) \leq \frac{sim_{Jac}(X,Y) \cdot \alpha + \beta}{\alpha - \beta} = sim_{Jac}(X, Y'_u)^{up}$$

where $\alpha = |X| + |Y|$, $\beta = (1 + sim_{Jac}(X,Y)) \cdot u$

Proof. By definition, we have

$$sim_{Jac}(X,Y) = \frac{\sum_{i=1}^{|\Sigma|} \min\{x_i, y_i\}}{\sum_{i=1}^{|\Sigma|} \max\{x_i, y_i\}}$$

and $\sum_{i=1}^{|\Sigma|} \max\{x_i, y_i\} = |X| + |Y| - \sum_{i=1}^{|\Sigma|} \min\{x_i, y_i\}$. Using the above two equations, we have

$$\sum_{i=1}^{|\Sigma|} \min\{x_i, y_i\} = \frac{(|X| + |Y|) \cdot sim_{Jac}(X,Y)}{sim_{Jac}(X,Y) + 1}.$$

Thus,

$$\begin{aligned}
& sim_{Jac}(X, Y'_u) \\
\leq & \frac{u + \sum_{i=1}^{|\Sigma|} \min\{x_i, y_i\}}{-u + \sum_{i=1}^{|\Sigma|} \max\{x_i, y_i\}} \\
= & \frac{u + \sum_{i=1}^{|\Sigma|} \min\{x_i, y_i\}}{-u + |X| + |Y| - \sum_{i=1}^{|\Sigma|} \min\{x_i, y_i\}} \\
= & \frac{u + \frac{(|X|+|Y|) \cdot sim_{Jac}(X,Y)}{sim_{Jac}(X,Y)+1}}{-u + |X| + |Y| - \frac{(|X|+|Y|) \cdot sim_{Jac}(X,Y)}{sim_{Jac}(X,Y)+1}} \\
= & \frac{(|X| + |Y| + u) \cdot sim_{Jac}(X,Y) + u}{|X| + |Y| - u \cdot sim_{Jac}(X,Y) - u}
\end{aligned}$$

Let $\alpha = |X| + |Y|$ and $\beta = (1 + sim_{Jac}(X,Y)) \cdot u$, the upper bound can be simplified as the following.

$$sim_{Jac}(X, Y'_u) \leq \frac{sim_{Jac}(X,Y) \cdot \alpha + \beta}{\alpha - \beta} = sim_{Jac}(X, Y'_u)^{up}$$

□

Example 4.1 (Computation of Progressive Upper Bound for Jaccard Similarity)**.** *Let us consider the same multisets* X *and* Y *as in Example 3.2. We have* $|X| = 5$, $|Y| = 6$, *and* $sim_{Jac}(X,Y) = 0.57$. *Suppose* $u = 1$, *the upper bound of the Jaccard similarity after an update is* $sim_{Jac\_up}(X, Y'_1) = 0.85$.

Table 4.1: Symbols Used in the Pruning-based Algorithms

Symbol	Interpretation
T_i	the i^{th} object in a database (DB)
m	number of objects
q_t	the query at time t that consists of the elements in the sliding window at time t
n	size of the sliding window, *i.e.*, query length
$sim(T_i, T_j)$	the exact similarity of two objects, T_i and T_j
$sim(T_i, T_j)_{up}$	the upper bound of similarity score of two objects, T_i and T_j
$\Delta sim(T_i)_+$	the maximum increase in similarity score regarding the next query
k	number of objects in the result
$topk_t$	the set of k objects that are most similar to the query at time t
$k^{th}best$	the k^{th} best result in the current result set
v_i	the estimated least number of updates needed for object T_i to enter $topk$
u_i	a number associated with object T_i indicating in how many steps the current bound for T_i would expire

4.2 A Pruning-based Algorithm (GP)

A brute-force method is to compute the similarity score between the updated query and each object at each time instant, and then sort the scores and get the top-k list. This approach is costly and involves heavy unnecessary computation, since when the query window slides only a small number of positions, e.g., 1 or 2, the changes of the similarity scores are limited. We present a heuristic algorithm that finds the exact top-k objects continuously, as shown in Algorithm 1. Table 4.1 lists the symbols used in the GP algorithm.

For clearer presentation of our method, let us assume that the arrival of new elements is synchronized with time, *i.e.*, a new element arrives at the stream whenever there is an update in time. The query at time $t+1$ shares a large portion of elements (at least $\frac{n-1}{n}$) with the query at time t. Thus, we do not expect a dramatic change in the top-k list, which means that the probability is low that the objects with small similarity scores at time t may enter the top-k list at time $t+1$. We can divide the objects into two categories according to their current similarity scores. In the first category, the similarity scores are small enough so that the objects will not enter the top-k list in the next several updates. The other objects that belong to the second category are regarded as candidates and need to be further verified

Algorithm 1: A General Pruning Algorithm (GP)

```
Input: $q_t$; $m$ objects; $topk_{t-1}$; similarity score regarding $q_{t-1}$ or its upper bound for
       each object; $u_i$ for each object.
Output: $topk_t$
1  foreach $T_i \in topk_{t-1}$ do
2      Compute $sim(T_i, q_t)$ and add to $topk_t$;
3  foreach $T_i \notin topk_{t-1}$ do
4      if $u_i \leq 1$ or $sim(T_i, q_t)_{up} > k^{th}best$ then
5          Compute $sim(T_i, q_t)$;
6          $u_i \leftarrow$ Null;
7          if $sim(T_i, q_t) > k^{th}best$ then
8              Update $topk_t$;
9      else
10         $T_i$ is pruned;
11         $u_i \leftarrow u_i - 1$;
12 foreach $T_i$ with $u_i = Null$ do
13     $est\_bound\big(sim(T_i, q_t), k^{th}best\big)$;
```

by computing the exact similarity scores.

For the objects in the first category, there is no need to compute the exact scores in the next time instant. We only need to compute their upper bounds at the next update. If the upper bound of an object is already smaller than the exact similarity scores between the new query and k other objects, then we can safely prune the object. To further reduce computation cost, we only compute bounds for a selected portion of the objects in each update. For each object T_i, we estimate v_i, the least number of updates needed for an object to enter the top-k list. We then pick a random integer u_i between $\frac{v_i}{2}$ and v_i. We store u_i and the upper bound after u_i updates. The computation details are described in a subroutine shown in Algorithm 2.

In each subsequent update, an object T_i can be pruned if its associated u_i is larger than one and its associated upper bound is no smaller than the k^{th} best score in the current top-k result. We simply decrease u_i by one since the upper bound associated with T_i would expire in $u_i - 1$ updates after the current update. For other objects that are regarded as candidates, we compute the exact similarity score. If the score is greater than the minimum score in the top-k list, this object enters the list and the object with the smallest score in the list leaves. After all the objects are processed once, for each object T_i whose exact similarity is computed at the current time instant, we reset u_i and compute the corresponding upper

Algorithm 2: Update u_i and Similarity Bounds after Re-computation of Similarity Scores

```
Function: $est\_bound(sim(T_i, q_t), k^{th}best)$
Input: $sim(T_i, q_t)$; $k^{th}best$ in $topk_t$;
Output: $u_i; sim(T_i, q_t)_{up}$.
1 if $sim(T_i, q_t) < k^{th}best$ then
2     Compute $\Delta sim(T_i)_+$;
3     $v_i \leftarrow \lceil \frac{k^{th}best - sim(T_i, q_t)}{\Delta sim(T_i)_+} \rceil$;
4     $u_i \leftarrow randint(\frac{v_i}{2}, v_i)$;
5     $sim(T_i, q_t)_{up} \leftarrow$ upper bound after $u_i$ updates;
6 else
7     $u_i \leftarrow 0$;
8     $sim(T_i, q_t)_{up} \leftarrow sim(T_i, q_t)$;
9 Output $u_i$, $sim(T_i, q_t)_{up}$;
```

bound. In fact, in each update, we need to scan through every object, either for decreasing u_i, or computing exact similarity. In our algorithm, if an object is said to be pruned in an update, we mean that there is no need to compute the exact similarity for this object regarding the new query.

In our implementation, we maintain a min-heap with k elements that stores the top-k objects for efficient updates. Suppose the elements in each object are sorted in a global order, the worst case time complexity of our algorithm is $\mathcal{O}(m \cdot \max_{1 \leq i \leq m} \{n, |T_i|\})$ where m is the number of objects, n is the query length, and $|T_i|$ is the length of the i^{th} object. This worst case happens when no object can be pruned in an update and we need to compute the exact similarity scores between the updated query and all the objects. The performance of our pruning-based method is highly related to the similarity score between the evolving query and the k^{th} best object in the top-k result. In each update, we do not need to compute the exact similarity scores for a large portion of the objects, especially when there are exactly k or slightly more than k objects with very high similarity scores while the other objects are very dissimilar to the query.

Finally, we discuss the exactness of this method. To ensure that our method can find the exact top-k objects in each update, we first compute the exact similarity scores between the updated query and each object in the top-k result of the previous query. The top-k list is updated accordingly. The other objects are then processed one by one. The top-k list always consists of the k objects with the largest similarity scores among the objects that have been processed so far. Since the minimum score in the top-k list does not decrease during this process, if the score of an object is smaller than the minimum score in the top-k list when it is processed, it must be smaller than the minimum score of the exact top-k list. Therefore, the algorithm always returns the exact top-k result.

4.3 Generalization to Other Similarity Measures

String similarity search is a fundamental operation in many applications such as data cleaning, information retrieval, and bioinformatics [15]. Edit distance is often used to measure the dissimilarity of strings.

Cosine similarity has been widely used for comparing the similarity among documents. A document can be represented by a vector where each component is the frequency of a particular word in the document. This type of term-frequency vector is used in applications including information retrieval, text document clustering, biological taxonomy, and gene feature mapping [24].

To efficiently apply the framework proposed in Section 4.2 to other similarity measures, we need to derive efficient and effective bounds for future updates based on current similarity/distance and its related functions. In this section, we show that the progressive bounds for edit distance, edit similarity, and cosine similarity can be computed in constant time.

4.3.1 Edit Distance and Edit Similarity

Definition 4.1 (Edit Distance and Edit Similarity)**.** *Given two strings X and Y of length m and n respectively, the edit distance $dis_{ed}(X, Y)$ is the minimum cost required to convert X to Y. Typically, we consider three edit operations, namely, insertion, deletion and substitution. The costs of three edit operations, denoted by δ_i, δ_d and δ_s, are non-negative and can be different. The edit similarity is defined as*

$$sim_{ed}(X, Y) = 1 - \frac{dis_{ed}(X, Y)}{\max\{\delta_i, \delta_d, \delta_s\} \cdot \max\{|X|, |Y|\}}.$$

Example 4.2 (Computation of Edit Similarity)**.** *Suppose the costs of insertion, deletion and substitution are 2, 1, and 3, respectively. $X = ababb$ and $Y = bbabac$ are two strings that consist of symbols in alphabet $\Sigma = \{a, b, c\}$. The edit distance table is shown in Table 4.2(a). From the table, we have $dis_{ed}(X, Y) = 8$. Thus, the edit similarity score between X and Y is $sim_{ed}(X, Y) = 1 - \frac{8}{3\times 6} = 0.44$.*

Table 4.2: Edit distance tables for Example 4.2 and Example 4.3

(a) $\delta_i = 2, \delta_d = 1, \delta_s = 3$

	ϵ	b	b	a	b	a	c
ϵ	0	2	4	6	8	10	12
a	1	3	5	4	6	8	10
b	2	1	3	5	4	6	8
a	3	2	4	3	5	4	6
b	4	3	2	4	3	5	7
b	5	4	3	5	4	6	8

(b) $\delta_i = \delta_d = \delta_s = 1$

	ϵ	b	b	a	b	a	c
ϵ	0	1	2	3	4	5	6
a	1	1	2	2	3	4	5
b	2	1	1	2	2	3	4
a	3	2	2	1	2	2	3
b	4	3	2	2	1	2	3
b	5	4	3	3	2	2	3

Since edit similarity is a function/transformation of edit distance, we first compute the upper and lower bounds for edit distance. We show in Property 4.2 that edit distance satisfies triangle inequality. Thus, we can derive the upper and lower bounds of edit distance using triangle inequality as shown in Property 4.3. Property 4.5 shows the bounds for edit similarity.

Property 4.2 (Triangle Inequality of Edit Distance [10]). *Given any three strings X, Y, and Z, the edit distance between X and Y satisfies the following triangle inequality.*

$$dis_{ed}(X,Y) \leq dis_{ed}(X,Z) + dis_{ed}(Z,Y).$$

Proof. If we want to transform string X into string Y, we can first transform X into another arbitrary string Z and then transform Z into Y. This kind of two-step edit operation may not have the cheapest cost. Thus, we have $dis_{ed}(X,Y) \leq dis_{ed}(X,Z) + dis_{ed}(Z,Y)$. That is, edit distance satisfies triangle inequality. As shown in Figure 4.1(a), we can think of XY to be the shortest path between X and Y. An alternative path that consists of two edges XZ and ZY can not have lower cost and this path achieves the same cost as the shortest path if and only if Z falls on XY. □

Property 4.3 (Progressive Bounds for Edit Distance with Unit Penalty Functions). *Let us consider edit distance with unit penalty functions where the cost for each deletion, insertion, and substitution operation is 1. Let X, Y be two strings and Y'_u be the string with u updates on Y. Given the edit distance $dis_{ed}(X,Y)$ between X and Y, without the knowledge of the updated elements in Y'_u, we have*

$$\max\{0, dis_{ed}(X,Y) - 2u\} \leq dis_{ed}(X,Y'_u) \leq dis_{ed}(X,Y) + 2u.$$

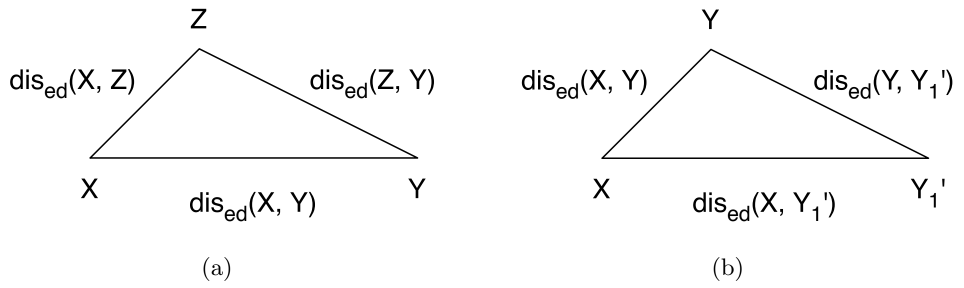


Figure 4.1: Graph Representations for Applying Triangle Inequality

Proof. By the definition of edit distance with unit penalty functions and how queries are updated, we have $dis_{ed}(Y, Y_1') \in \{0, 1, 2\}$. Let us consider the following triangle inequalities on a triangle with vertices X, Y, and Y_1', as shown in Figure 4.1(b).

$$dis_{ed}(X, Y_1') \leq dis_{ed}(X, Y) + dis_{ed}(Y, Y_1') \tag{4.1}$$

$$dis_{ed}(X, Y_1') \geq |dis_{ed}(X, Y) - dis_{ed}(Y, Y_1')| \tag{4.2}$$

From Inequality 4.1, 4.2, and the range of $dis_{ed}(Y, Y_1')$, we have

$$\max\{0, dis_{ed}(X, Y) - 2\} \leq dis_{ed}(X, Y_1') \leq dis_{ed}(X, Y) + 2. \tag{4.3}$$

More generally, for edit distance after u updates, we can apply Inequality 4.3 u times iteratively and get the bounds as stated in the property. □

Property 4.4 (Progressive Bounds for Edit Distance with General Penalty Functions)**.** *Let us consider edit distance with general penalty functions as discussed in Definition 4.1, where the penalties for insertion, deletion and substitution are denoted as δ_i, δ_d, and δ_s, respectively. Let X, Y be two strings and Y_u' be the string with u updates on Y. Given the edit distance $dis_{ed}(X, Y)$ between X and Y, without the knowledge of the updated elements in Y_u', we have*

$$\max\{0, dis_{ed}(X, Y) - u \cdot \max\{\delta_i + \delta_d, \delta_s\}\} \leq dis_{ed}(X, Y_u') \leq dis_{ed}(X, Y) + u \cdot \max\{\delta_i + \delta_d, \delta_s\}.$$

Proof. By the definition of edit distance with general penalty functions and how queries are updated, we have

$$0 \leq dis_{ed}(Y, Y_1') \leq \max\{\delta_i + \delta_d, \delta_s\}.$$

Let us also apply triangle inequalities on a triangle with vertices X, Y, and Y_1', as shown in Figure 4.1(b). Recalling Inequality 4.1 and 4.2 and considering the range of $dis_{ed}(Y, Y_1')$, we have the following bounds.

$$\max\{0, dis_{ed}(X,Y) - \max\{\delta_i + \delta_d, \delta_s\}\} \leq dis_{ed}(X, Y_1') \leq dis_{ed}(X,Y) + \max\{\delta_i + \delta_d, \delta_s\}. \tag{4.4}$$

More generally, for edit distance after u updates, we can apply Inequality 4.4 u times iteratively and get the bounds as stated in the property. □

Property 4.5 (Progressive Bounds for Edit Similarity with Unit Penalty Functions)**.** *Let us consider edit distance with unit penalty functions where the cost for each deletion, insertion, and substitution operation is* 1*. Let X, Y be two strings and Y_u' be the string with u updates on Y. Given the edit similarity $sim_{ed}(X,Y)$ between X and Y, without the knowledge of the updated elements in Y_u', we have*

$$sim_{ed}(X,Y) - \frac{2u}{\max\{|X|,|Y|\}} \leq sim_{ed}(X, Y_u') \leq \min\{1, sim_{ed}(X,Y) + \frac{2u}{\max\{|X|,|Y|\}}\}.$$

Proof. By the definition of edit similarity and Property 4.3, we have

$$\begin{aligned} sim_{ed}(X, Y_u') &\leq 1 - \frac{\max\{0, dis_{ed}(X,Y) - 2u\}}{\max\{|X|,|Y|\}} \\ &= \min\{1, sim_{ed}(X,Y) + \frac{2u}{\max\{|X|,|Y|\}}\} \\ sim_{ed}(X, Y_u') &\geq 1 - \frac{dis_{ed}(X,Y) + 2u}{\max\{|X|,|Y|\}} \\ &= sim_{ed}(X,Y) - \frac{2u}{\max\{|X|,|Y|\}} \end{aligned}$$

□

Property 4.6 (Progressive Bounds for Edit Similarity with General Penalty Functions)**.** *Let us consider edit distance with general penalty functions as discussed in Definition 4.1, where the penalties for insertion, deletion and substitution are denoted as δ_i, δ_d, and δ_s, respectively. Let X, Y be two strings and Y_u' be the string with u updates on Y. Given the edit similarity $sim_{ed}(X,Y)$ between X and Y, without the knowledge of the updated elements in Y_u', we have*

$$sim_{ed}(X, Y_u') \geq sim_{ed}(X,Y) - \frac{u \cdot \max\{\delta_i + \delta_d, \delta_s\}}{\max\{\delta_i, \delta_d, \delta_s\} \cdot \max\{|X|,|Y|\}} \tag{4.5}$$

$$sim_{ed}(X, Y_u') \leq \min\{1, sim_{ed}(X,Y) + \frac{u \cdot \max\{\delta_i + \delta_d, \delta_s\}}{\max\{\delta_i, \delta_d, \delta_s\} \cdot \max\{|X|,|Y|\}}\}. \tag{4.6}$$

Proof. By the definition of edit similarity with general penalty function and Property 4.4, we have

$$\begin{aligned}
sim_{ed}(X, Y'_u) &\leq 1 - \frac{\max\{0, dis_{ed}(X,Y) - u \cdot \max\{\delta_i + \delta_d, \delta_s\}\}}{\max\{\delta_i, \delta_d, \delta_s\} \cdot \max\{|X|, |Y|\}} \\
&= \min\{1, sim_{ed}(X,Y) + \frac{u \cdot \max\{\delta_i + \delta_d, \delta_s\}}{\max\{\delta_i, \delta_d, \delta_s\} \cdot \max\{|X|, |Y|\}}\} \\
sim_{ed}(X, Y'_u) &\geq 1 - \frac{dis_{ed}(X,Y) + u \cdot \max\{\delta_i + \delta_d, \delta_s\}}{\max\{\delta_i, \delta_d, \delta_s\} \cdot \max\{|X|, |Y|\}} \\
&= sim_{ed}(X,Y) - \frac{u \cdot \max\{\delta_i + \delta_d, \delta_s\}}{\max\{\delta_i, \delta_d, \delta_s\} \cdot \max\{|X|, |Y|\}}
\end{aligned}$$

□

Example 4.3 (Progressive Bounds for Edit Similarity)**.** *Let us consider the same strings X and Y as in Example 4.2. If we use unit penalty functions, the edit distance between X and Y is* 3*, as shown in Table 4.2(b). The edit similarity is* $1 - \frac{3}{1*6} = 0.50$*. Suppose* $u = 1$*, we have* $1 \leq dis_{ed}(X, Y'_u) \leq 5$ *and* $0.167 \leq sim_{ed}(X, Y'_u) \leq 0.833$*. Let us then consider the general penalty functions where* $\delta_i = 2, \delta_d = 1$ *and* $\delta_s = 3$*. As shown in Table 4.2(a), the edit distance between X and Y is* 8*. Thus, the edit similarity is* $1 - \frac{8}{3*6} = \frac{5}{9} = 0.56$*. Suppose* $u = 1$*, we have* $5 \leq dis_{ed}(X, Y'_u) \leq 11$ *and* $0.39 \leq sim_{ed}(X, Y'_u) \leq 0.72$*.*

4.3.2 Cosine Similarity

Definition 4.2 (Cosine Similarity)**.** *Let us consider the vector representation mentioned in Chapter 3. Given two vectors, $\vec{X}$ and $\vec{Y}$, the cosine similarity is defined using a dot product and magnitudes of vectors.*

$$sim_{cos}(\vec{X}, \vec{Y}) = \frac{\vec{X} \cdot \vec{Y}}{\|\vec{X}\|\|\vec{Y}\|} = \frac{\sum_{i=1}^{|\Sigma|} \vec{X}_{(i)} \times \vec{Y}_{(i)}}{\sqrt{\sum_{i=1}^{|\Sigma|} (\vec{X}_{(i)})^2} \times \sqrt{\sum_{i=1}^{|\Sigma|} (\vec{Y}_{(i)})^2}}$$

where $\vec{X}_{(i)}$ and $\vec{Y}_{(i)}$ are the i-th components of vectors $\vec{X}$ and $\vec{Y}$, respectively.

Example 4.4 (Computation of Cosine Similarity)**.** *Given an alphabet* $\Sigma = \{e_1, e_2, e_3, e_4, e_5\}$*,* $X = \{e_3, e_2, e_2, e_1, e_5, e_2, e_1\}$ *and* $Y = \{e_1, e_2, e_1, e_3, e_5, e_4, e_2, e_3, e_4\}$ *are two multisets over* Σ*. We can transform X and Y to vectors, that is,* $\vec{X} = \{2, 3, 1, 0, 1\}$ *and* $\vec{Y} = \{2, 2, 2, 2, 1\}$*. We have* $\vec{X} \cdot \vec{Y} = 13$*,* $\|\vec{X}\| = \sqrt{15}$ *and* $\|\vec{Y}\| = \sqrt{17}$*. Thus,* $sim_{cos}(\vec{X}, \vec{Y}) = \frac{13}{\sqrt{15}*\sqrt{17}} = 0.81$*.*

Given the exact cosine similarity, dot product value of two vectors, and statistics on the current query, we can estimate the cosine similarity after a number of updates in constant time. The following property holds for computing the upper and lower bounds for the updated similarity score.

Property 4.7 (Progressive Bounds for Cosine Similarity)**.** *Let us consider two multi-sets X and Y and their corresponding vector representations $\vec{X}$ and $\vec{Y}$. If the exact cosine similarity between $\vec{X}$ and $\vec{Y}$ is $sim_{cos}(\vec{X},\vec{Y})$, the dot product value is $\vec{X}\cdot\vec{Y}$, and the value of the maximum component of $\vec{X}$ is $h_{\vec{X}}$, then we have the following bounds for cosine similarity after u updates. Note that we use notations $|Y|$, the cardinality of multi-set Y, and $\sum_{i=1}^{|\Sigma|}\vec{Y}_{(i)}$ interchangeably.*

$$sim_{cos}(\vec{X},\vec{Y}'_u) \geq \begin{cases} \frac{1}{\alpha}\cdot(1-\phi)\cdot sim_{cos}(\vec{X},\vec{Y}) & ,\vec{X}\cdot\vec{Y}\neq 0 \\ 0 & ,\vec{X}\cdot\vec{Y}=0 \end{cases}$$

$$sim_{cos}(\vec{X},\vec{Y}'_u) \leq \begin{cases} \frac{1}{\beta}\cdot(1+\phi)\cdot sim_{cos}(\vec{X},\vec{Y}) & ,\vec{X}\cdot\vec{Y}\neq 0 \\ \frac{h_{\vec{X}}\cdot u}{\beta\|\vec{X}\|\|\vec{Y}\|} & ,\vec{X}\cdot\vec{Y}=0 \end{cases}$$

where

$$\alpha = \sqrt{(2u+1)-\frac{u^2+u}{|Y|}}$$

$$\beta = \frac{|Y|}{\sqrt{(|Y|-u)^2+u}}$$

and

$$\phi = \frac{h_{\vec{X}}\cdot u}{\vec{X}\cdot\vec{Y}} \quad ,\vec{X}\cdot\vec{Y}\neq 0$$

Proof. Let $\vec{Y}'_u$ be the updated vector of $\vec{Y}$ after u updates. Two inequalities used in this proof are as follows.

$$|Y| \leq \|\vec{Y}\|^2 \leq |Y|^2 \tag{4.7}$$

$$\sqrt{\|\vec{Y}\|^2+u-|Y|^2+(|Y|-u)^2} \leq \|\vec{Y}'_u\| \leq \sqrt{\|\vec{Y}\|^2-u+|Y|^2-(|Y|-u)^2} \tag{4.8}$$

We first prove the lower bound using the above inequalities.

When $\vec{X} \cdot \vec{Y} \neq 0$,

$$\begin{aligned}
sim_{cos}(\vec{X}, \vec{Y'_u}) &= \frac{\vec{X} \cdot \vec{Y'_u}}{\|\vec{X}\| \|\vec{Y'_u}\|} \geq \frac{(\vec{X} \cdot \vec{Y'_u})_{min}}{(\|\vec{X}\| \|\vec{Y'_u}\|)_{max}} \\
&= \frac{\vec{X} \cdot \vec{Y} - h_{\vec{X}} \cdot u}{\|\vec{X}\| \cdot \sqrt{\|\vec{Y}\|^2 - u + |Y|^2 - (|Y| - u)^2}} \\
&= \frac{\vec{X} \cdot \vec{Y} - h_{\vec{X}} \cdot u}{\frac{\vec{X} \cdot \vec{Y}}{sim_{cos}(\vec{X}, \vec{Y}) \cdot \|\vec{Y}\|} \cdot \sqrt{\|\vec{Y}\|^2 + 2u \cdot |Y| - u^2 - u}} \\
&= \frac{\|\vec{Y}\|}{\sqrt{\|\vec{Y}\|^2 + 2u \cdot |Y| - u^2 - u}} (1 - \frac{h_{\vec{X}} \cdot u}{\vec{X} \cdot \vec{Y}}) sim_{cos}(\vec{X}, \vec{Y}) \\
&\geq \frac{1}{\sqrt{1 + \frac{2u \cdot |Y| - u^2 - u}{|Y|}}} (1 - \frac{h_{\vec{X}} \cdot u}{\vec{X} \cdot \vec{Y}}) sim_{cos}(\vec{X}, \vec{Y}) \\
&= \frac{1}{\alpha} (1 - \frac{h_{\vec{X}} \cdot u}{\vec{X} \cdot \vec{Y}}) sim_{cos}(X, Y)
\end{aligned}$$

where $\alpha = \sqrt{(2u + 1) - \frac{u^2 + u}{|Y|}}$.

When $\vec{X} \cdot \vec{Y} = 0$,

$$sim_{cos}(\vec{X}, \vec{Y'_u}) = \frac{\vec{X} \cdot \vec{Y'_u}}{\|\vec{X}\| \|\vec{Y'_u}\|} \geq \frac{(\vec{X} \cdot \vec{Y'_u})_{min}}{(\|\vec{X}\| \|\vec{Y'_u}\|)_{max}} = 0$$

The upper bound can be derived similarly.

When $\vec{X} \cdot \vec{Y} \neq 0$,

$$\begin{aligned}
sim_{cos}(\vec{X}, \vec{Y'_u}) &= \frac{\vec{X} \cdot \vec{Y'_u}}{\|\vec{X}\|\|\vec{Y'_u}\|} \leq \frac{(\vec{X} \cdot \vec{Y'_u})_{max}}{(\|\vec{X}\|\|\vec{Y'_u}\|)_{min}} \\
&= \frac{\vec{X} \cdot \vec{Y} + h_{\vec{X}} \cdot u}{\|\vec{X}\| \cdot \sqrt{\|\vec{Y}\|^2 + u - |Y|^2 + (|Y| - u)^2}} \\
&= \frac{\vec{X} \cdot \vec{Y} + h_{\vec{X}} \cdot u}{\frac{\vec{X} \cdot \vec{Y}}{sim_{cos}(\vec{X}, \vec{Y}) \cdot \|\vec{Y}\|} \cdot \sqrt{\|\vec{Y}\|^2 - 2u \cdot |Y| + u^2 + u}} \\
&= \frac{\|\vec{Y}\|}{\sqrt{\|\vec{Y}\|^2 - 2u \cdot |Y| + u^2 + u}}(1 + \frac{h_{\vec{X}} \cdot u}{\vec{X} \cdot \vec{Y}}) sim_{cos}(\vec{X}, \vec{Y}) \\
&\leq \frac{1}{\sqrt{1 + \frac{u^2 + u - 2u \cdot |Y|}{|Y|^2}}}(1 + \frac{h_{\vec{X}} \cdot u}{\vec{X} \cdot \vec{Y}}) sim_{cos}(\vec{X}, \vec{Y}) \\
&= \frac{1}{\beta}(1 + \frac{h_{\vec{X}} \cdot u}{\vec{X} \cdot \vec{Y}}) sim_{cos}(\vec{X}, \vec{Y})
\end{aligned}$$

where $\beta = \frac{|Y|}{\sqrt{(|Y| - u)^2 + u}}$.

When $\vec{X} \cdot \vec{Y} = 0$,

$$\begin{aligned}
sim_{cos}(\vec{X}, \vec{Y'_u}) &= \frac{\vec{X} \cdot \vec{Y'_u}}{\|\vec{X}\|\|\vec{Y'_u}\|} \leq \frac{(\vec{X} \cdot \vec{Y'_u})_{max}}{(\|\vec{X}\|\|\vec{Y'_u}\|)_{min}} \\
&= \frac{\vec{X} \cdot \vec{Y} + h_{\vec{X}} \cdot u}{\|\vec{X}\| \cdot \sqrt{\|\vec{Y}\|^2 + u - |Y|^2 + (|Y| - u)^2}} \\
&\leq \frac{h_{\vec{X}} \cdot u}{\|\vec{X}\|\|\vec{Y}\|\sqrt{1 + \frac{u^2 + u - 2u \cdot |Y|}{|Y|^2}}} \\
&= \frac{h_{\vec{X}} \cdot u}{\beta\|\vec{X}\|\|\vec{Y}\|}
\end{aligned}$$

□

Example 4.5 (Computation of Progressive Bounds for Cosine Similarity)**.** *Let us consider the same vectors in Example 4.4. We have* $\vec{X} \cdot \vec{Y} = 13$, $h_{\vec{X}} = 3$, $|Y| = 9$ *and* $sim_{cos}(\vec{X}, \vec{Y}) = 0.81$. *Suppose* $u = 1$, *the upper bound of the cosine similarity after an update is* $sim_{cos\_up}(\vec{X}, \vec{Y'_1}) = \frac{\sqrt{65}}{9} * (1 + \frac{3}{13}) * 0.81 = 0.89$ *and the corresponding lower bound is* $sim_{cos\_lo}(\vec{X}, \vec{Y'_1}) = \frac{3}{5} * (1 - \frac{3}{13}) * 0.81 = 0.37$.

Our pruning-based method supports the similarity measures defined above and can be further extended to similarity/distance measures, such as hamming distance, overlap similarity and dice similarity, whose progressive bounds can be computed in constant time.

Chapter 5

A MinHash-based Method

In this chapter, we use the MinHash [6] technique, a locality-sensitive hashing (LSH) scheme, to approximate Jaccard similarity. We design indices for efficient updating estimated similarity scores. Since the MinHash technique is designed for estimating sets rather than multi-sets, we slightly change the definition of the current query to the set of distinct elements in the current sliding window.

5.1 Top-k Similarity Search for Static Queries

We first discuss how to estimate Jaccard similarity using MinHash for static objects and queries.

Definition 5.1 (Image under Permutation [1])**.** *Let $[n]$ denote the set $\{0, \ldots, n-1\}$. A permutation π on $[n]$ is a bijective function (one-to-one correspondence) from $[n]$ to itself. If $x \in [n]$, then $\pi(x)$, the value of π when applied to x, is the image of x under π. The image of a subset $X \subseteq [n]$ under π is defined as follows.*

$$\pi[X] = \{y \mid y = \pi(x) \textit{ for each } x \in X\}$$

Definition 5.2 (Min-Wise Independent Permutations [7])**.** *Let S_n be the set of all permutations of $[n]$. A family of permutations $\mathcal{F} \subseteq S_n$ is min-wise independent if for any set $X \subseteq [n]$ and any element $x \in X$, we have*

$$Pr(\min\{\pi[X]\} = \pi(x)) = \frac{1}{|X|}$$

where the permutation π is chosen randomly in F, $\pi[X]$ is the image of X under π, and $\pi(x)$ is the image of x under π.

Property 5.1 (Jaccard Similarity Estimation [27]). *Let us consider a query q and an arbitrary object T_i in which the elements are drawn from a lexicon, Σ. Let h be a hash function that maps elements in Σ to distinct integers in range $[0, |\Sigma| - 1]$ and is randomly picked from a family of min-wise independent permutations as discussed in Definition 5.2. Also, let the MinHash value, $\min(h(T_i))$, be the element in T_i with the smallest hash value. Jaccard similarity between T_i and q can be estimated using the following equation.*

$$Pr[\min(h(T_i)) = \min(h(q))] = \frac{|T_i \cap q|}{|T_i \cup q|}$$

The MinHash values for all objects and that of the query can be stored in a MinHash signature matrix, M, where each entry $M(i, j)$ is the MinHash value of the j^{th} itemset under hash function h_i. Based on Property 5.1, the Jaccard similarity between two itemsets can be estimated by the ratio of the number of rows containing the same MinHash values to the number of all the rows in the signature matrix. The detailed computation is shown in Example 5.1.

Example 5.1 (Jaccard Similarity Estimation). *We consider the matrix representation of four objects and a static query q shown in Table 5.1(a). Each column in the matrix represents a set and an element is in the set if the corresponding entry is 1. We apply five random permutations defined in Table 5.1(b), as the min-wise independent hash functions, on objects and the query and get the signature matrix shown in Table 5.1(c). For example, since object T_1 contains element a, b, and c, the corresponding hash values according to h_4 are 1, 3, and 4, respectively. The MinHash value of T_1 according to h_4 is a since a is the element in T_1 with the smallest hash value.*

Then, we can estimate the Jaccard similarity score between two sets by computing the ratio of the number of rows containing the same MinHash values to the number of random permutations. For example, the MinHash values of T_1 and q are the same according to random permutations h_2 and h_3. Thus, the estimated Jaccard similarity between T_1 and q is $\frac{2}{5}$. We compare the estimated Jaccard similarity with the exact Jaccard similarity for each object in Table 5.1(d). When objects are ordered in descending estimated similarity score regarding the query, we have $T_3 > T_1 = T_4 > T_2$, which is identical to the order using exact similarity. Therefore, in this example, we can get the top-k result accurately using estimated scores.

Table 5.1: An Example of Jaccard Similarity Estimation

(a) Matrix Representation of Sets

Element	T_1	T_2	T_3	T_4	q
a	1	1	0	1	0
b	1	0	1	0	1
c	1	0	1	0	1
d	0	1	0	1	1
e	0	0	1	1	1

(b) Random Permutations

Element	h_1	h_2	h_3	h_4	h_5
a	1	2	3	1	0
b	2	3	0	3	4
c	3	0	1	4	3
d	4	4	2	0	1
e	0	1	4	2	2

(c) Signature Matrix for Example 5.1

	T_1	T_2	T_3	T_4	q
h_1	a	a	e	e	e
h_2	c	a	c	e	c
h_3	b	d	b	d	b
h_4	a	d	e	d	d
h_5	a	a	e	a	d

(d) Comparison on Exact and Estimated Jaccard Similarity

$sim_{Jac}(T_i, q)$	T_1	T_2	T_3	T_4
Estimated	0.40	0.20	0.60	0.40
Exact	0.40	0.25	0.75	0.40

5.2 Top-k Similarity Search over Evolving Queries

To answer evolving queries, given a fixed set of hash functions, the MinHash values for all the objects are fixed and only the MinHash values for the query are subject to modification. That is, we only need to update the MinHash values for the query. Therefore, we have a straightforward solution shown in Algorithm 3. Table 5.2 lists the symbols used in the MinHash-based algorithms for ease of presentation. Algorithm 3 first computes the MinHash signature for the updated query. To determine if we shall update the estimated Jaccard similarity of object T_j regarding the updated query q_t according to a hash function h_i, we consider the following 4 cases based on whether the MinHash values of the object and the queries are equal or not.

- Case 1: If $mh_i(T_j) \neq mh_i(q_{t-1})$ and $mh_i(T_j) = mh_i(q_t)$, the estimated Jaccard similarity between T_j and the updated query should be increased by $\frac{1}{l}$ where l is the

Table 5.2: Notations used by MinHash-based Algorithms

Notation	Interpretation
$mh(q_t)$	A list of MinHash values of q_t according to l hash functions.
$mh_i(q_t)$	The MinHash value of q_t according to the i^{th} hash function, h_i, $i \in [1, l]$.
S_{T_j}	The estimated similarity score between T_j and the current query, $j \in [1, m]$.
I_i	Inverted index built on the MinHash values of the i^{th} hash function, $i \in [1, l]$.
$I_i(v)$	Inverted list of hash value v which consists of the objects whose MinHash value is v according to the i^{th} hash function, $i \in [1, l]$ and $v \in [1, \lvert\Sigma\rvert]$.

Algorithm 3: A MinHash-based Algorithm (MHB)

```
Input: $q_t$; $l$ hash functions; $mh(q_{t-1})$; MinHash table for $m$ objects.
Output: $topk_t$
1 Compute $mh(q_t)$;
2 foreach $j \in \{1, ..., m\}$ do
3     foreach $i \in \{1, ..., l\}$ do
4         if $mh_i(T_j) \neq mh_i(q_{t-1})$ and $mh_i(T_j) = mh_i(q_t)$ then
5             $S_{T_j} \leftarrow S_{T_j} + \frac{1}{l}$;
6         else if $mh_i(T_j) = mh_i(q_{t-1})$ and $mh_i(T_j) \neq mh_i(q_t)$ then
7             $S_{T_j} \leftarrow S_{T_j} - \frac{1}{l}$;
8     if $T_j \notin topk_t$ and $S_{T_j} > \min_{heap\_element}$ then
9         Update $topk_t$;
```

number of hash functions applied.

- Case 2: If $mh_i(T_j) = mh_i(q_{t-1})$ and $mh_i(T_j) \neq mh_i(q_t)$, the estimated Jaccard similarity between T_j and the updated query should be decreased by $\frac{1}{l}$.
- Case 3: If $mh_i(T_j) = mh_i(q_{t-1})$ and $mh_i(T_j) = mh_i(q_t)$, we do not change the estimated Jaccard similarity between T_j and the updated query.
- Case 4: If $mh_i(T_j) \neq mh_i(q_{t-1})$ and $mh_i(T_j) \neq mh_i(q_t)$, we do not change the estimated Jaccard similarity between T_j and the updated query.

Thus, for each hash function, the estimated Jaccard similarity of an object T_j regarding the updated query changes only when Case 1 and Case 2 occur. The time complexity of

Figure 5.1: Inverted Indices for MinHash Values

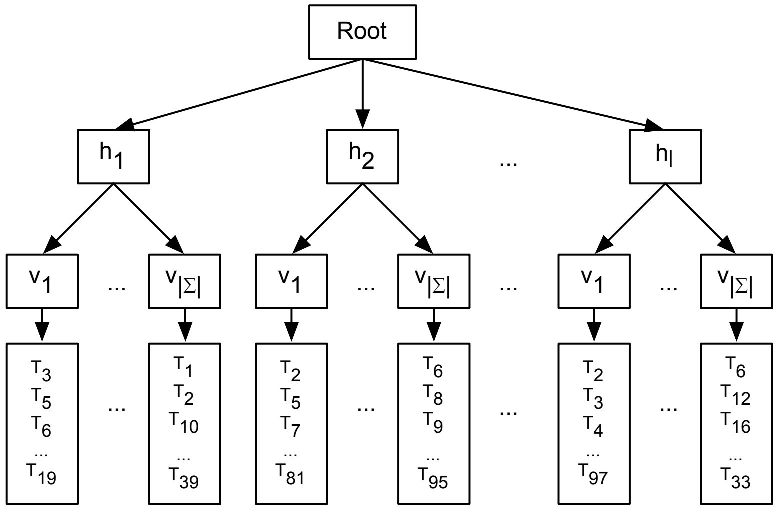


Table 5.3: An Example of Inverted Indices for MinHash Values

(a) Index for h_1

$a : T_1, T_2$
$e : T_3, T_4$

(b) Index for h_3

$b : T_1, T_3$
$d : T_2, T_4$

(c) Index for h_5

$a : T_1, T_2, T_4$
$e : T_3$

(d) Index for h_2

$a : T_2$
$c : T_1, T_3$
$e : T_4$

(e) Index for h_4

$a : T_1$
$d : T_2, T_4$
$e : T_3$

this algorithm is $O(l \cdot m)$, where m is the number of objects and l is the number of hash functions used for estimating the similarity scores.

We can further reduce the computational cost using inverted indices. The MinHash signature matrix that stores the MinHash values of objects can be transformed into l inverted indices whose structure is shown in Figure 5.1. For each hash function, its corresponding inverted index stores a mapping from each existing MinHash value to a list of object ids, *i.e.*, inverted list. Moreover, elements in each inverted list are sorted in ascending order of object id, which enable more efficient list merge and intersection operations. In this way, we can efficiently retrieve the objects with a specified MinHash value according to a certain hash function.

Algorithm 4: A MinHash-based Algorithm using Inverted Indices (MHI)

```
  Input: $q_t$; $l$ hash functions; $mh(q_{t-1})$; inverted indices.
  Output: $topk_t$
1 Compute $mh(q_t)$;
2 foreach $i \in \{1, ..., l\}$ do
3     if $mh_i(q_{t-1}) \neq mh_i(q_t)$ then
4         for $T_j \in I_i(mh_i(q_t))$ do
5             $S_{T_j} \leftarrow S_{T_j} + \frac{1}{l}$;
6         for $T_j \in I_i(mh_i(q_{t-1}))$ do
7             $S_{T_j} \leftarrow S_{T_j} - \frac{1}{l}$;
8 Find $topk_t$ using updated scores;
```

Example 5.2 (Inverted Indices for MinHash Values)**.** *Let us consider the objects in Example 5.1. The inverted indices for all* 5 *hash functions are shown in Table 5.3.*

An algorithm that uses the inverted indices is presented in Algorithm 4. The MinHash signature for the updated query is computed firstly. We need to search the inverted index of a hash function h_i only when the MinHash values of q_t and q_{t-1} regarding h_i are different. Suppose the MinHash value of the updated query has changed according to h_i, to determine which objects' similarity scores to update, we recall Case 1 and Case 2 mentioned earlier in this section. Each case corresponds to a scan in an inverted list of h_i. To be more precise, corresponding to Case 1, we increase the estimated Jaccard similarity by $\frac{1}{l}$ for objects in the inverted list of hash value $mh_i(q_t)$. That is, instead of checking for each object as in Algorithm 3, we search for the objects that satisfy Case 1 utilizing the index. Similarly, we decrease the estimated Jaccard similarity by $\frac{1}{l}$ for objects in the inverted list of hash value $mh_i(q_{t-1})$, which corresponds to Case 2.

The worst case of this algorithm is $O(l \cdot m)$, where m is the number of objects and l is the number of hash functions. It is achieved when the MinHash values of consecutive queries differ under all the hash functions. However, this case could hardly occur because the Jaccard similarity between consecutive queries is in fact very high. On the other extreme, the best case of this algorithm is $O(l)$ when the MinHash signature of the current query is exactly the same as that of the previous query.

Example 5.3 (Updating Top-k Result using MHI)**.** *Let us continue the example for estimating Jaccard similarity discussed in Example 5.1. Suppose $k = 1$, the top-*1 *object is T_3 regarding query $q_{t-1} = \{c, d, b, e\}$. At time t, the query is updated to $q_t = \{d, b, e, a\}$. The*

Table 5.4: Comparison on Exact and Estimated Jaccard Similarity for Example 5.3

$sim_{Jac}(T_i, q_t)$	T_1	T_2	T_3	T_4
Estimated	0.80	0.20	0.60	0.40
Exact	0.75	0.20	0.75	0.60

MinHash signature lists for q_{t-1} *and* q_t *are* $[e, c, b, d, d]$ *and* $[e, c, b, a, a]$*, respectively. Based on Algorithm 4, we only need to search the inverted index of* h_4 *and* h_5 *since the MinHash signatures of* q_{t-1} *and* q_t *only differ in* 2 *positions.*

Using the inverted lists computed in Example 5.2, for h_4*, we decrease the estimated similarity score by* 0.20 *for objects in* $I_4(d) = \{T_2, T_4\}$ *and increase the estimated similarity score by* 0.20 *for objects in* $I_4(a) = \{T_1\}$*. We follow the same procedure for* h_5*. The estimated Jaccard similarity regarding the updated query is shown in Table 5.4. The top-*1 *object suggested by Algorithm 4 becomes* T_1*.*

Chapter 6

Experiments

To validate the effectiveness and efficiency of our two methods named GP and MHI, we first report the experimental results with respect to Jaccard similarity on synthetic data sets in Section 6.1. The results on two real data sets on market basket data and click stream data are presented in Section 6.2. We also show the performance of the pruning algorithms on Cosine similarity and edit distance in Section 6.3 using the click stream data set. All methods are implemented in Python and all experiments are conducted on a PC computer with Intel Core 2 Duo E8400 3.00GHz CPU and 8GB main memory running 64-bit Microsoft Windows 7.

6.1 Results on Synthetic Data Sets

The following results are generated using the IBM Quest data generator[1]. We conduct experiments to test the efficiency and accuracy of our methods regarding the following parameters.

- k - top-k;
- n - number of items per query/ average number of items per object.
- $|\Sigma|$ - lexicon size;
- m - number of objects;

[1]http://www.cs.loyola.edu/~cgiannel/assoc_gen.html

- l - number of different hash functions/ random permutations.

To generate synthetic datasets using the IBM Quest data generator, we set parameters for n, $|\Sigma|$, and m. The synthetic query stream is generated by concatenating random objects whose lengths are between $0.8\bar{n}$ to $1.2\bar{n}$, where $\bar{n}$ is the average object length of the data set. In reality, the distribution of the elements in the stream would not change much in a very short time period. For example, on a Q&A forum, some hot topics and its related ones may be discussed heavily for an hour before moving to the next groups of hot topics. In terms of average querying time, comparing to the real scenario, the performance of our algorithms using the generated query stream may be slower. It is because in our way of generating the stream, the transit from one topic to another is relatively faster than the typical real case.

We use the variable controlling method to conduct our experiments. The values of controlled variables along with the corresponding figures for each test are shown in Table 6.1. In each variable controlled test, we compare the performance of GP and MHI, on the average querying time with two baseline methods, namely, BFM and MHB. BFM is a brute-force method that computes the exact similarity scores for every object with respect to a query. MHB is a baseline method based on MinHash technique but without indexing structures. To better illustrate how our upper bounds derived in Chapter 4 can be used to prune unpromising objects, we define the pruning effectiveness of an update in Definition 6.1. In our experiments, we report the average pruning effectiveness of hundreds of updates.

Definition 6.1 (Pruning Effectiveness of Pruning-based Method)**.** *Suppose we have m objects in total and we compute the exact similarity scores for m^* objects during an update. The pruning effectiveness of this update is $1 - \frac{m^*}{m}$.*

The *accuracy* in our context is defined in Definition 6.2. The pruning algorithm GP always reports the exact query results, which has 100% accuracy. However, since the MinHash-based methods compute similarity scores approximately, MHB and MHI give estimated answers to the top-k query. We report the average accuracy of MinHash methods in Figure 6.2.

Definition 6.2 (Accuracy of a Method)**.** *Suppose the exact top-k list regarding query q_t is top_k^e and the k-th best object is denoted as $o_{r_k}^e$. Given a method A that returns a top-k list, top_k^A, regarding query q_t, the accuracy of method A is the proportion of objects in top_k^A whose exact similarity scores regarding q_t is no smaller than the k-th best similarity score*

Table 6.1: Values of Controlled Variables for Tests on Jaccard Similarity

Test	k	n	$\|\Sigma\|$	l	m	Figures
Synthetic Data (Vary k)	n/a	10	$10,000$	200	$10,000$	Fig. 6.1(a), 6.2(a)
Synthetic Data (Vary k)	n/a	10	20	200	$10,000$	Fig. 6.1(b), 6.2(b)
Synthetic Data (Vary $\|\Sigma\|$)	10	10	n/a	200	$10,000$	Fig. 6.1(c), 6.2(c)
Synthetic Data (Vary n)	10	n/a	$10,000$	200	$10,000$	Fig. 6.1(d), 6.2(d)
Synthetic Data (Vary l)	10	10	$10,000$	n/a	$10,000$	Fig. 6.1(e), 6.2(e)
Synthetic Data (Vary m)	10	10	$10,000$	200	n/a	Fig. 6.1(f), 6.2(f)
Market Basket (Vary l)	10	10	$16,470$	n/a	$88,162$	Fig. 6.4(a), 6.4(b)
Market Basket (Vary k)	n/a	10	$16,470$	200	$88,162$	Fig. 6.4(c), 6.4(d)
Click Stream (Vary l)	10	5	17	n/a	$31,790$	Fig. 6.5(a), 6.5(b)
Click Stream (Vary k)	n/a	5	17	50	$31,790$	Fig. 6.5(c), 6.5(d)

in top_k^e. More formally, acc_A, the accuracy of method A for q_t is defined as

$$acc_A = \frac{|\{o|o \in top_k^A \wedge sim(o, q_t) \geq sim(o^e_{r_k}, q_t)\}|}{k}.$$

6.1.1 Efficiency

The average querying time of the four methods when k varies is shown in Figures 6.1(a) and 6.1(b) for two synthetic data sets with large lexicon (10^4) and small lexicon (20), respectively. We can observe that in both cases, the two baseline methods almost have no change in average processing time and MHI that uses 200 hash functions outperforms the other three methods greatly. The pruning effectiveness of GP is shown in Figure 6.3(a) and Figure 6.3(b) when the lexicon size is set to 10^4 and 20, respectively. GP's pruning power drops gradually because the similarity score of the k-th best object in the top-k result tends to decrease when k increases. Given other parameters fixed, the similarity score of the k-th best object in the result with respect to queries would be smaller. Thus, the pruning effectiveness generally is weaker in the case of larger lexicon size.

Figure 6.1(c) provides a better illustration of the trends of different methods when the lexicon size changes. When the lexicon size increases by orders of magnitude, we can see that the pruning effectiveness drops from 59.4% to 44.5% gradually. Thus, the average processing time of GP increases. The average running time of MHB and BFM does not have obvious change with respect to lexicon size. However, the average running time of MHI first decreases and then increases. When the lexicon size is small, the length of each inverted list is relatively large. Thus, a change in MinHash value of the query may result in many updates in the estimated similarity scores. When lexicon size becomes very large, though the size of each inverted list is small, the number of unmatched MinHash values between the MinHash lists of the previous query and the new query increases, which results in more queries on the inverted indices.

We also examine how the average query answering time changes with respect to average object length. The results on efficiency and pruning effectiveness are shown in Figures 6.1(d), and 6.3(c). The average processing time of BFM increases linearly while the performance of MHB does not have obvious change with respect to average object length. It is because we have transformed the transctions of varied length into MinHash signatures of the same length. However, the average processing time of MHI first decreases slightly and then increases. When the average object length becomes larger, the size of each inverted list would be larger, which results in longer processing time. When the average object length increases, the pruning effectiveness of GP increases dramatically, from 46.8% to 97.2%. However, its average running time still increases slightly, since there is an increase in computing exact similarity scores for larger sets in the verification phase.

The results with respect to the number of hash functions is shown in Figure 6.1(e). The average processing time of MHB and MHI both increases linearly with the number of hash functions, which is in accordance with the analysis of our algorithms. The scalability of three methods is shown in Figure 6.1(f). All the methods increases linearly as the number of object increases. The pruning power of GP increases from 19.1% to 64.0% when the number of objects increases from 10^3 to 10^5, which is shown in Figure 6.3(e).

6.1.2 Accuracy

We also test the accuracy of the MinHash-based algorithms. By our definition, the accuracy of the two MinHash-based algorithms are the same according to the same set of hash functions. We denote the MinHash methods by MH in the figures that report accuracy. The accuracy can be affected by two factors, the number of hash functions used and the intrinsic characteristics of our data set such as the distribution of similarity scores among objects.

In general, we can achieve an accuracy ranging from 70% to 98% when 200 hash functions are used. Figures 6.2(a) and 6.2(b) show the change in accuracy when k increases for data

Table 6.2: Real Data Sets Statistics

Data Set	Cardinality	Avg. Length	Lexicon Size
Market Basket	88,162	10.306	16,470
MSNBC	31,790	5.3338	17

sets with large lexicon and small lexicon, respectively. In the case where lexicon size is 10^4, the accuracy first decreases and then increases. The smallest average accuracy rate is achieved when $k = 10^2$. When k is very large, say 10^3, the k-th best similarity score would become 0, which is the reason why the accuracy increases to 100% when k is 10^3. This issue does not occur in the case of small lexicon. When lexicon size is 20, we can achieve accuracy in the range of 80% to 88%. The average accuracy first decreases and then increases, and the smallest average accuracy rate is achieved when $k = 10$.

The average accuracy increases quickly and then decreases slightly when the lexicon size increases, which is shown in Figure 6.2(c). The highest accuracy is achieved when lexicon size is 10^4. In Figure 6.2(d), when the average object length increases from 10 to 10^3, the accuracy drops drastically from 94% to 14.4%. The reason is apparent. More hash functions are needed to achieve the same level of accuracy when the set size gets larger.

We also test the trend of accuracy when the number of hash functions and the number of objects increase. The results are shown in Figures 6.2(e) and 6.2(f), respectively. When more hash functions are used, the estimated Jaccard similarity is closer to the exact score, thus lead to higher accuracy. As shown in Figure 6.2(e), we can see that the result follows this trend and we can achieve average accuracy rate ranging from 91% to 97.5%. Figure 6.2(f) suggests that the average accuracy first increases and then decreases when the number of objects increases. The highest accuracy is achieved when $m = 10^4$.

6.2 Results on Real Data Sets

We conduct experimental studies on two real data sets: a retail market basket data set[2] from an anonymous Belgian retail store and MSNBC, a data set of click-stream data. We use a subset of the MSNBC data set that removes the shortest sequences, with the link provided

[2] http://fimi.ua.ac.be/data/

[3]. We then transform each sequence into set. There is a significant difference in lexicon size of the two real data sets. The market basket data set has $16,470$ distinct items while the MSNBC data set contains only 17 distinct items. Table 6.2 shows the detailed statistics of the data sets. To generate the querying stream, we randomly concatenate objects in the data set whose size is between $0.8\bar{n}$ to $1.2\bar{n}$, where $\bar{n}$ is the average object length of the data set. For each data set, we compare the average processing time and the average accuracy when k or l varies, respectively. The results are shown in Figure 6.4 and 6.5.

The trends of the curves for the market basket data set is highly consistent with the results on the synthetic data sets. In Figure 6.4(a) and 6.4(b), we show the trends of processing time and accuracy of the MinHash based methods when different number of hash functions is used. The average processing time of MHB and MHI both increases linearly with the number of hash functions. The performance of MHI is around 50 to 65 times faster than MHB. The average accuracy increases steadily from 74% to 93% when the number of hash functions increases from 100 to 500. To compare the average processing time of different methods, we choose the default number of hash functions as 200. The processing time of BFM, MHB, and MHI does not have obvious change and MHI always has the best performance. The pruning power of GP decreases from 60% to 40% gradually when k increases. Therefore, the processing time of GP increases when k increases. The MinHash based algorithms can achieve accuracy over 80% when k is varied from 1 to 1000, which is shown in Figure 6.4(d).

The results for the click stream data set is shown in Figure 6.5. Since the average length of the objects in this data set is only 5.33, the MinHash based algorithms do not need many hash functions in order to achieve very high accuracy. As shown in Figure 6.5(b), the average accuracy is above 90% when over 30 hash functions are used. To test the preformance when k varies, we use 30 hash functions. MHI is still the best method with respect to average query answering time, which can outperform the BFM by more than an order of magnitude. Comparing to the results on market basket data set, the average accuracy of MinHash-based methods is generally higher while the pruning effectiveness of GP is lower for the click stream data set.

[3]http://www.philippe-fournier-viger.com/spmf/

6.3 Results on Other Similarity Measures

In this section, we experiment on the performance of the bounds, derived in Section 4.3, for cosine similarity and edit similarity, using the click stream data set. Since cosine similarity can be approximated using a random projection method [8], we can use the hashing-based framework for Jaccard similarity only with the signature generation phase changed. The random projection method chooses random hyperplanes defined by normal unit vectors and uses the signs of the dot products between the input vector and random hyperplanes as the signature for this vector. Given the signatures of two vectors generated by the same set of random hyperplanes, the percentage of matching bits in their signatures is proportional to the cosine distance between two vectors. Corresponding to MHB and MHI for Jaccard similarity, the hashing-based methods for cosine similarity are denoted as RPB and RPI. By default, we use 200 random hyperplanes in our experiment.

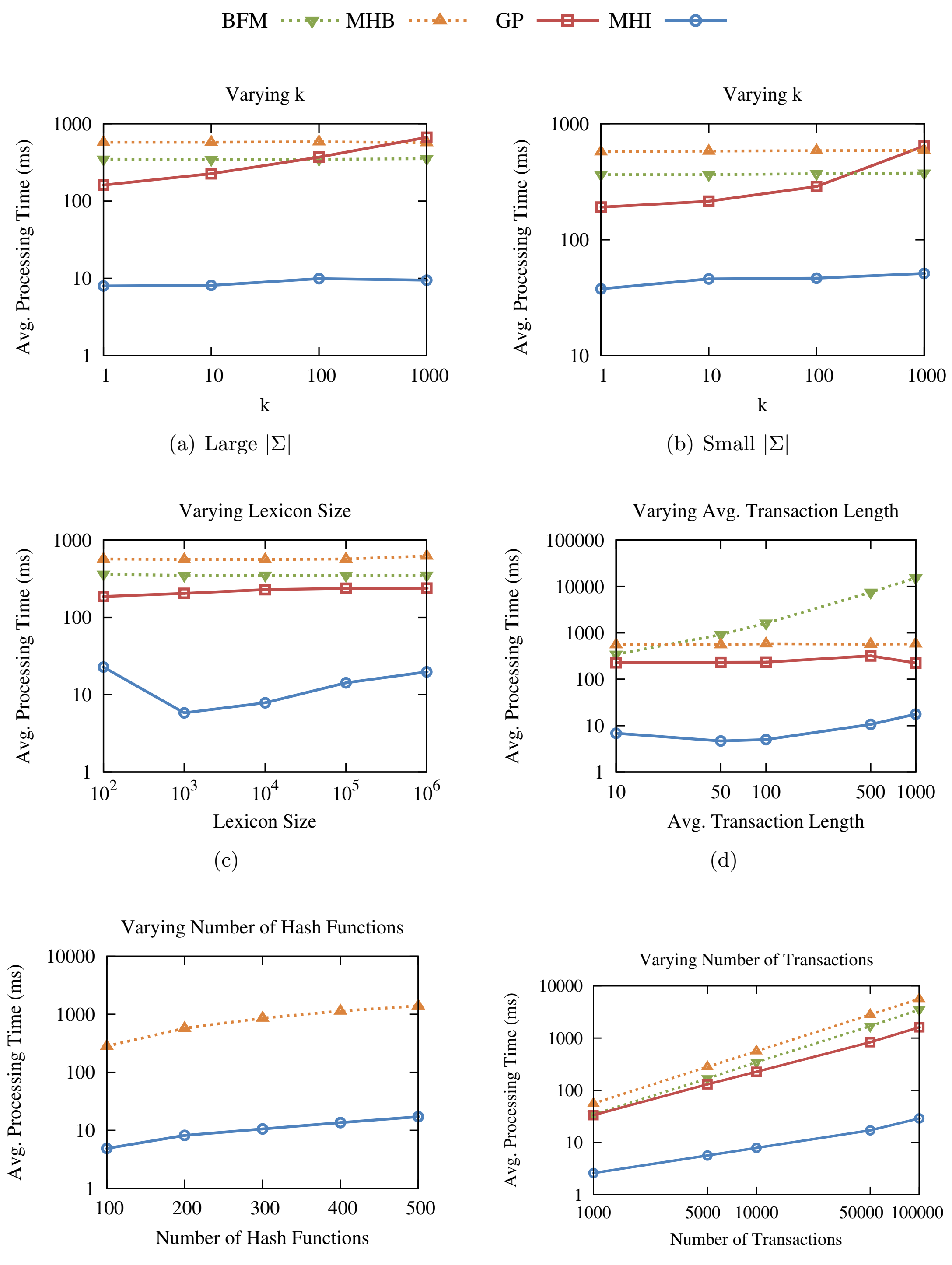
The results for cosine similarity is shown in Figure 6.6. The pruning effectiveness decreases from 86.0% to 78.6% when k varies from 1 to 1000. It is apparant since our similarity bound depends on the k^{th} best score. When k increases, the k^{th} best score decreases, thus less objects can be pruned. The average processing time increases when k increases, which is in accordance with the trend of the pruning effectiveness. The pruning-based method has better performance, in terms of both querying time and accuracy, than the hashing-based method. We should notice that signatures constructed by random projection method only consist of 0 and 1. Since we use inverted indices in the same way as for Jaccard similarity, the number of objects in each inverted list would be very large. It can be the reason why the performance of the hashing-based algorithm becomes worse for cosine similarity.

Figure 6.7 shows the results for edit similarity on click stream data set. The trend is similar to that of cosine similarity and Jaccard similarity. The algorithm can prune 64.8% to 46.8% objects when k increases from 1 to 100. Among the three distance measures, the bound for cosine similarity has the best pruning effectiveness; The bound for edit similarity comes second and the bound for Jaccard similarity is slightly worse than that of edit similarity.

6.4 Summary of Results

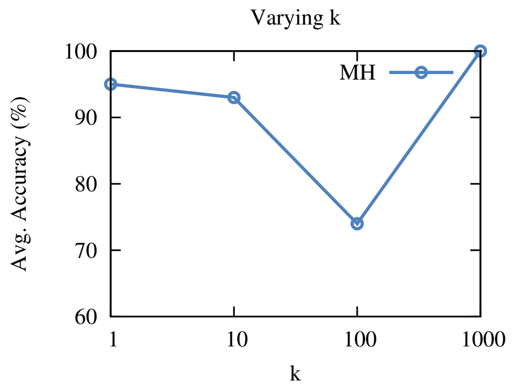
We conduct experiments to test the effectiveness and efficiency of our pruning-based method and hashing-based method. There are 5 parameters that can affect the performance of our methods. We test our methods by varying one parameter in each test and compare the performance of our methods.

Generally speaking, in terms of accuracy, the pruning-based method is an exact method while the hashing-based method can reach very good accuracy rate when a few hundreds of hash functions are used. For Jaccard similarity, the hashing-based method with good accuracy results can always achieve faster average querying time than the pruning-based method. However, for cosine similarity, the pruning method can achieve better performance than the hashing-based method. One reason is that the pruning effectiveness of cosine similarity is the highest among the three similarity measures in our experiments. The other reason is that the inverted indices can not reduce much work in updating the similarity scores since each inverted list can be very long with length equals to half of the number of transactions on average.

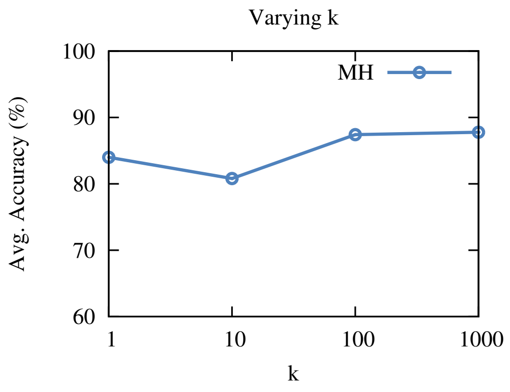


(a) Large $|\Sigma|$ (b) Small $|\Sigma|$ (c) (d) (e) (f)

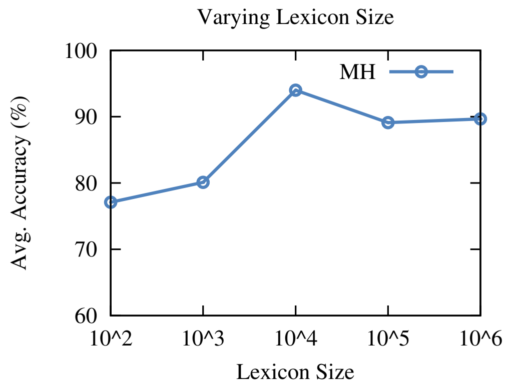
Figure 6.1: Average Processing Time on IBM Quest Data Set Using Jaccard Similarity



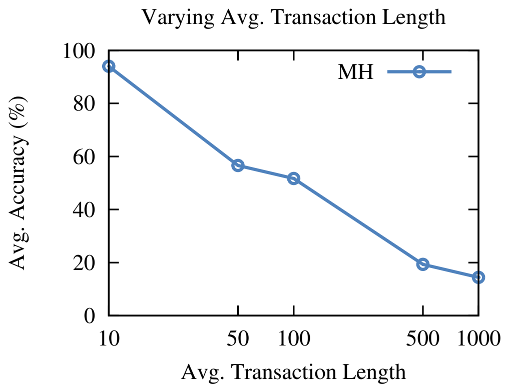
(a) Large $|\Sigma|$



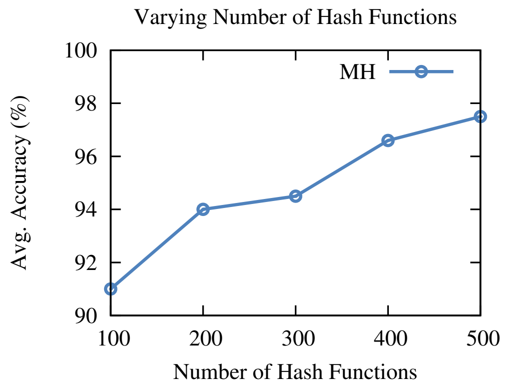
(b) Small $|\Sigma|$



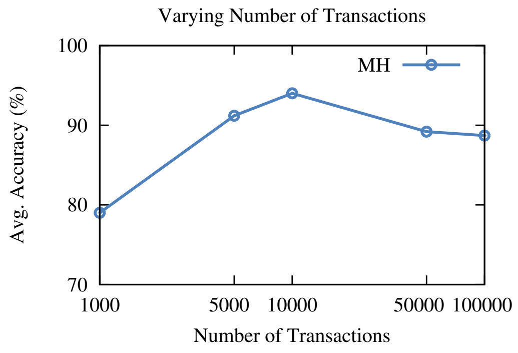
(c)



(d)

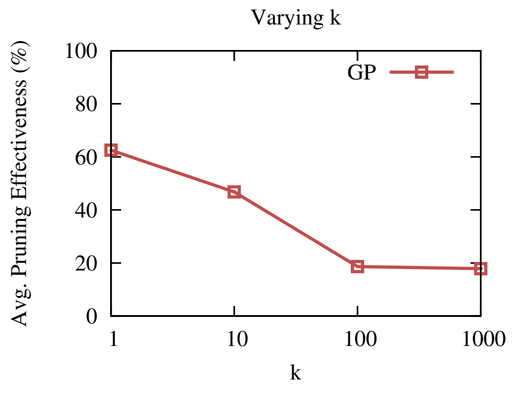


(e)

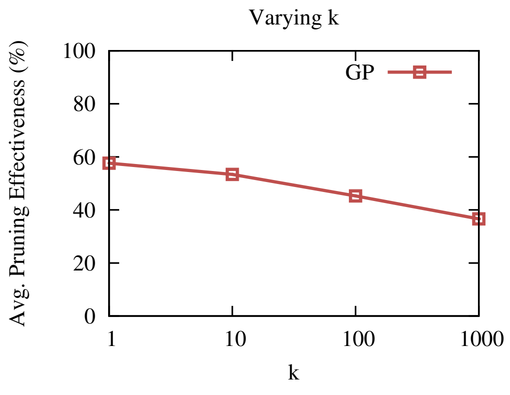


(f)

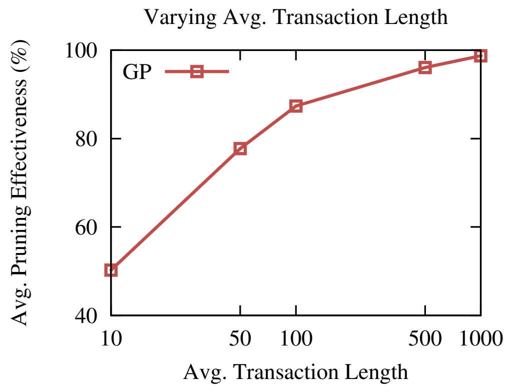
Figure 6.2: Average Accuracy of Hashing-based Method on IBM Quest Data Set Using Jaccard Similarity



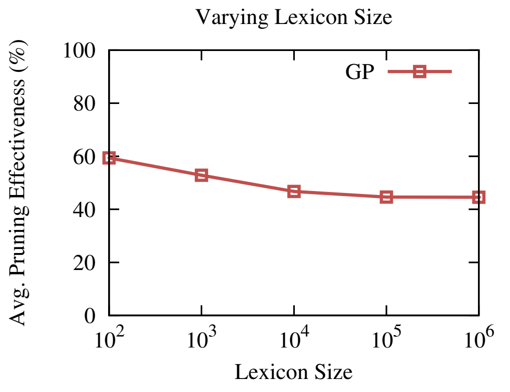
(a) Large $|\Sigma|$



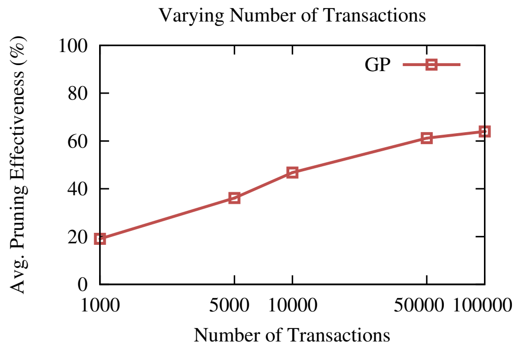
(b) Small $|\Sigma|$



(c)



(d)



(e)

Figure 6.3: Pruning Effectiveness of GP on IBM Quest Data Set Using Jaccard Similarity

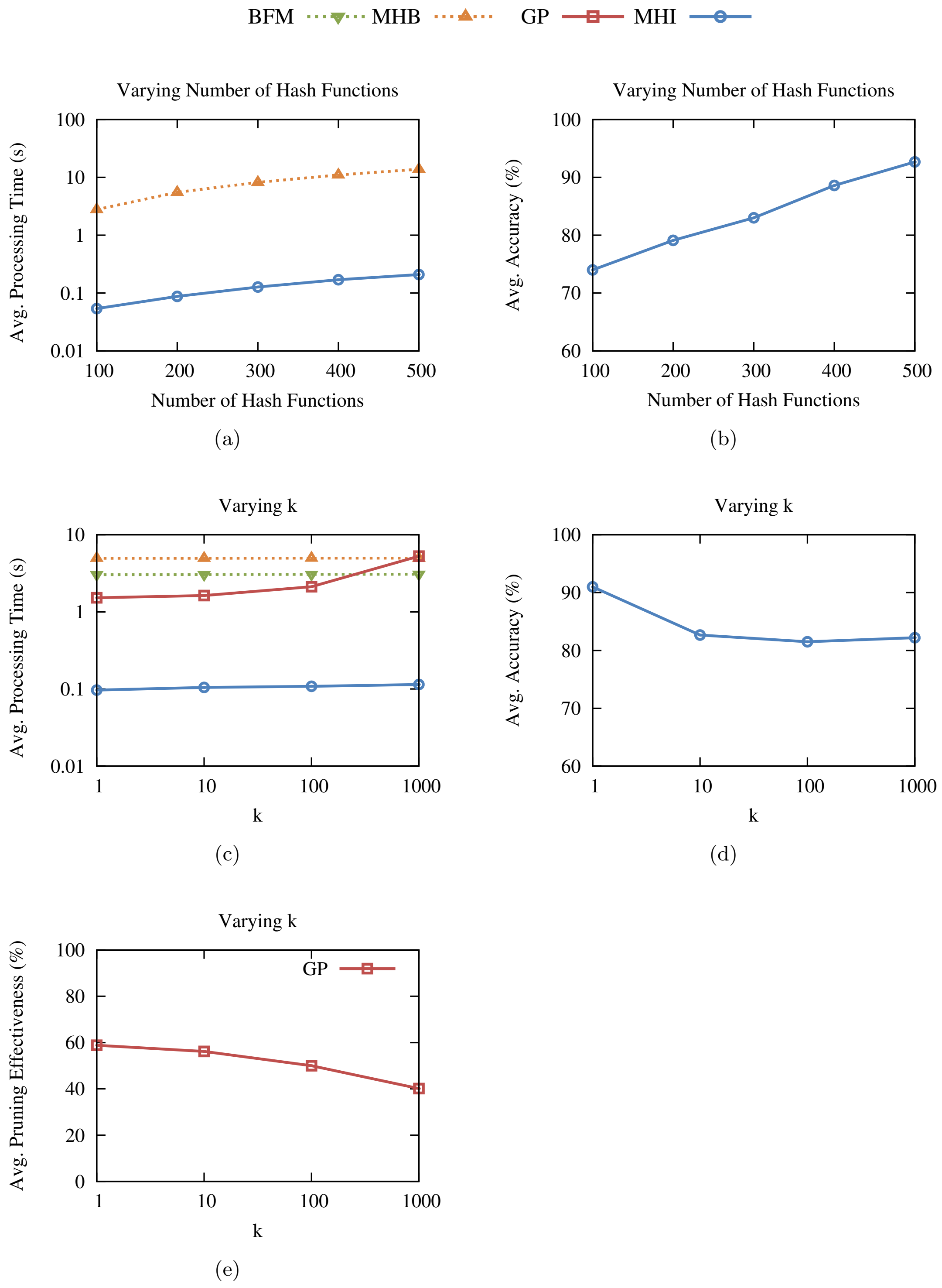


Figure 6.4: Results on Market Basket Dataset Using Jaccard Similarity

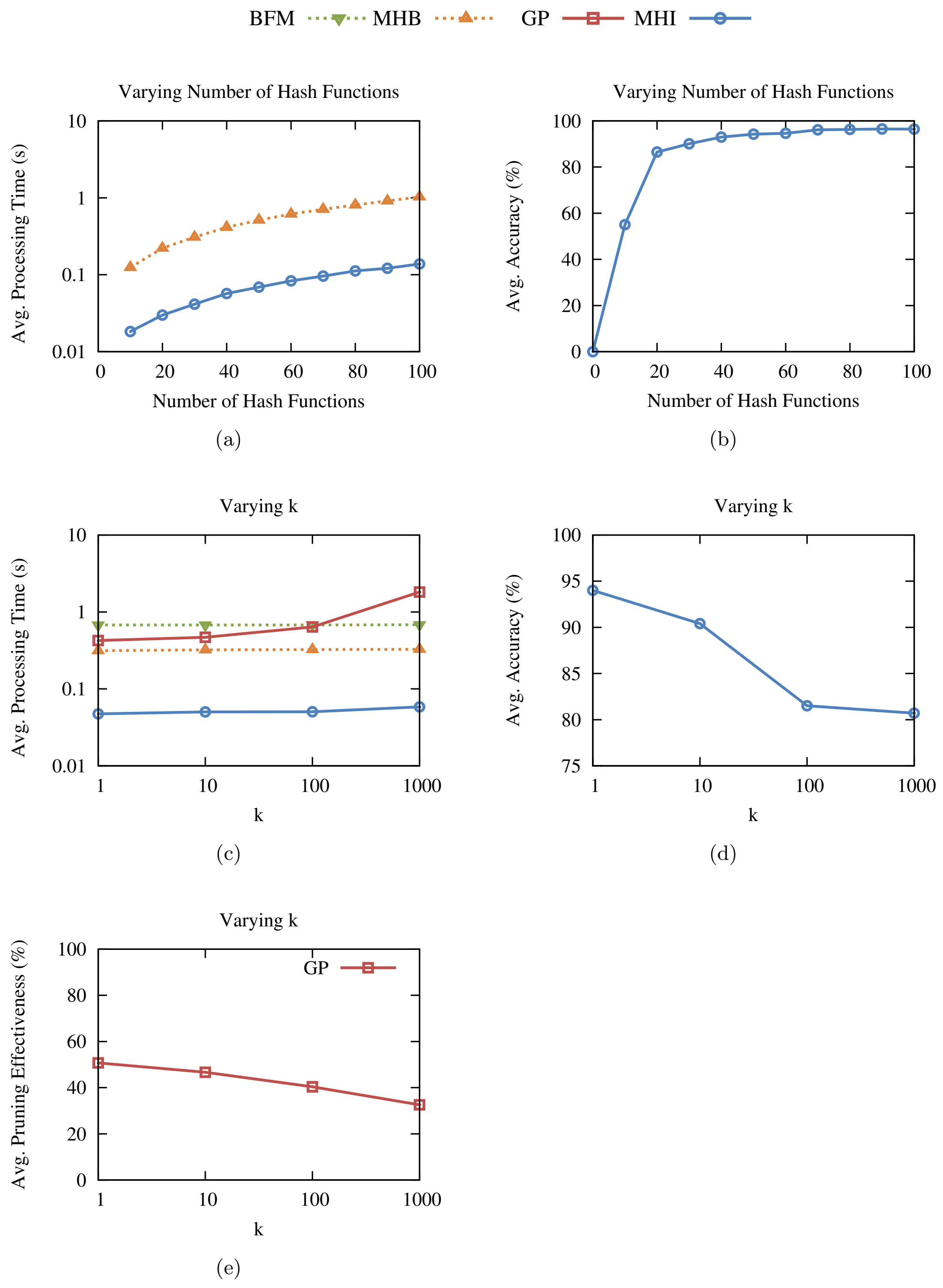
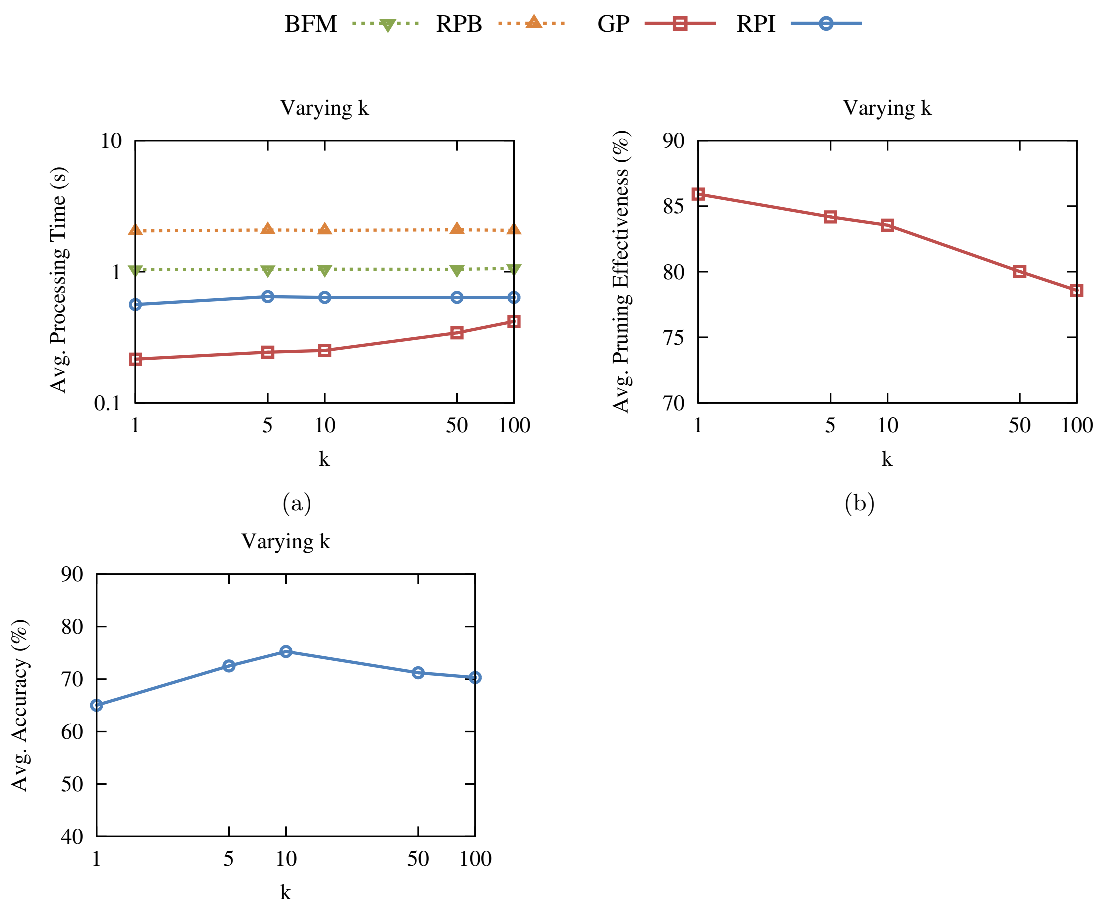


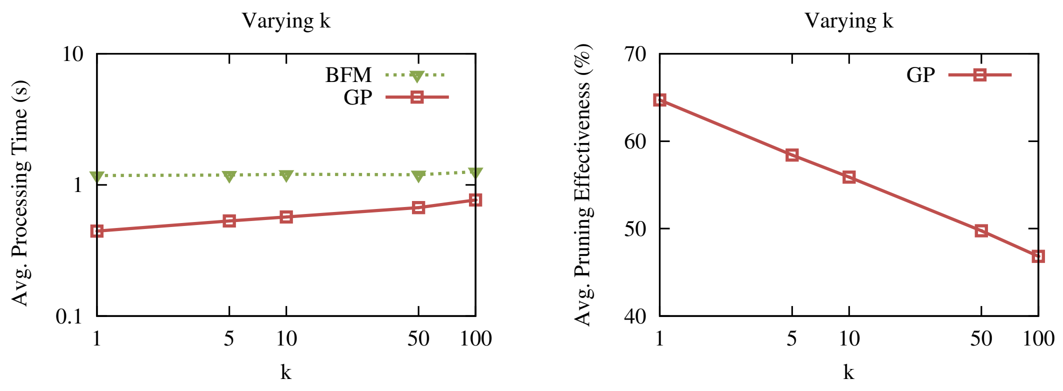
Figure 6.5: Results on Click Stream Dataset Using Jaccard Similarity



(a) (b)

(c)

Figure 6.6: Results on Click Stream Dataset Using Cosine Similarity



(a) (b)

Figure 6.7: Results on Click Stream Dataset Using Edit Similarity

Chapter 7

Conclusions

Motivated by a variety of application scenarios including advertisement recommendation for new users, automatic suggestions on computer war game strategies, and music recommendation by user humming, in this thesis, we studied the problem of *continuous similarity search for evolving queries*, which is to continuously find the top-k objects in a collection of sets that are most similar to an evolving query. To the best of our knowledge, it is the first research endeavour on this problem. We devised two efficient methods with different frameworks. The pruning-based method uses pruning strategies to reduce the cost of computing the exact similarity scores. This method can be extended to multiple similarity measures including edit distance and cosine similarity. The MinHash-based method approximates Jaccard similarity based on MinHash and efficiently updates the estimated scores using indexing structures.

We evaluated our methods empirically using synthetic and real data sets. Our synthetic data sets are generated using the IBM Quest Data Generator and the real data sets are market basket data and click stream data. The experimental results verify the effectiveness and efficiency of our methods. Moreover, the results on the real data sets are highly consistent with those on synthetic data sets.

As for future work, we can consider the following interesting directions.

- *Enhance the pruning effectiveness.* In our pruning-based method, we only consider the progressive bounds that result from the evolvement of our query. We may find a way to incorporate the evolving feature into the pruning techniques used in static similarity join and search algorithms to further prune candidates.

- *Improve hashing-based method for other similarity measures.* MinHash is an *locality-sensitive hashing (LSH)* scheme for Jaccard similarity. It not only provides an efficient way to estimate Jaccard similarity, but also performs as signatures that can be indexed for similarity comparison against evolving signatures. We can extend our hashing-based framework to other similarity/distance measures that can be approximated by an LSH scheme. For example, hamming distance can be approximated using a bit sampling scheme [28] and cosine similarity can be approximated using a random projection method [8]. However, as some preliminary results in Section 6.3 shows, the indexing structure is not good enough when every element in a signature can only take binary values. We may consider improving the inverted indices or constructing other indexing structures for this kind of similarity measures.

- *Similarity search over multiple evolving streams.* Let us consider a variation of our problem. Given a static object and multiple data streams with a fixed sliding window size, we want to continuously find the top-k data streams whose last n items is most similar to the static object. The progressive bounds derived in the pruning-based method still holds. A similar pruning algorithm can be devised based on the bounds. As for the hashing-based framework, unlike our problem that stores hash values for each static transaction and only update the hash values of the query, we need to update the hash values for each sliding window. Therefore, efficient techniques for updating or estimating the hash values is desired.

Bibliography

[1] Image (mathematics). http://en.wikipedia.org/wiki/Image_(mathematics). 28

[2] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006. 7

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International World Wide Web Conference*, pages 131–140, Banff, Canada, 2007. 3, 6

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. 7

[5] C. Böhm, B. C. Ooi, C. Plant, and Y. Yan. Efficiently processing continuous k-nn queries on data streams. In *ICDE*, pages 156–165, 2007. 9

[6] A. Z. Broder. On the resemblance and containment of documents. In *In Compression and Complexity of Sequences (SEQUENCES97*, pages 21–29. IEEE Computer Society, 1997. 4, 7, 28

[7] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:327–336, 1998. 28

[8] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 380–388, 2002. 7, 41, 50

[9] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006. 5

[10] L. Chen and R. T. Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004. 6, 20

[11] W. W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD International Conference on Magagement of Data*, pages 201–212, Seattle, WA, 1998. 3

[12] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–67, 1993. 3

[13] M. Datar and S. Muthukrishnan. Estimating rarity and similarity over data stream windows. In *ESA*, pages 323–334, 2002. 7

[14] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013. 7

[15] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013. 19

[16] L. Devroye and T. J. Wagner. Nearest neighbor methods in discrimination. *Handbook of Statistics*, 2, 1982. 3

[17] C. Faloutsos, R. Barber, M. Flickner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3:231–262, 1994. 3

[18] C. Faloutsos and D. W. Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, Department of Computer Science, University of Maryland, College Park, 1995. 3

[19] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the qbic system. *IEEE Computer*, 28:23–32, 1995. 3

[20] A. Gersho and R. M. Gray. *Vector Quantization and Data Compression.* Kluwer, 1991. 3

[21] A. Gionis, P. Indyk, and M. Rajeev. Similarity search in high dimensions via hashing. In *Proceedings of the International Conference on Very Large Data Bases*, 1999. 7

[22] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001. 6

[23] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984. 7

[24] J. Han, M. Kamber, and J. Pei. *Data Mining: Concepts and Techniques, 3rd ed.* Morgan Kaufmann, 2011. 19

[25] T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification. In *Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining*, pages 142–149, 1995. 3

[26] M. R. Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 284–291, Seattle, WA, 2006. 3

[27] P. Indyk. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, 38(1):84–90, 2001. 29

[28] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998. 7, 50

[29] V. Koivune and S. Kassam. Nearest neighbor filters for multivariate data. In *IEEE Workshop on Nonlinear Signal and Image Processing*, 1995. 3

[30] M. Kontaki and A. N. Papadopoulos. Efficient similarity search in streaming time sequences. In *SSDBM*, pages 63–72, 2004. 9

[31] N. Koudas, B. C. Ooi, K. Tan, and R. Zhang. Approximate nn queries on streams with guaranteed error/performance bounds. In *VLDB*, pages 804–815, 2004. 9

[32] G. Li, D. Deng, and J. Feng. A partition-based method for string similarity joins with edit-distance constraints. *ACM Trans. Database Syst.*, 38(2):9, 2013. 6

[33] X. Lian, L. Chen, and B. Wang. Approximate similarity search over multiple stream time series. In *DASFAA*, pages 962–968, 2007. 9

[34] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *SIGMOD Conference*, pages 635–646, 2006. 9

[35] K. Mouratidis and D. Papadias. Continuous nearest neighbor queries over sliding windows. *IEEE Trans. Knowl. Data Eng.*, 19(6):789–803, 2007. 9

[36] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE Trans. Knowl. Data Eng.*, 17(11):1451–1464, 2005. 9

[37] S. Muthukrishnan. *Data Streams: Algorithms and Applications.* Foundations and trends in theoretical computer science. Now Publishers, 2005. 8

[38] S. Pan and X. Zhu. Continuous top-k query for graph streams. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Managemen*, 2012. 9

[39] A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: tools for content-based manipulation of image databases. In *In Proceedings of the SPIE Conference on Storage and Retrieval of Image and Video Databases II*, 1994. 3

[40] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, New York, NY, 1983. 3

[41] A. W. M. Smeulders and R. Jain. *Image Databases and Multi-media Search.* World Scientific Pub Co Inc, 1997. 3

[42] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012. 6

[43] W. E. Winkler. The state of record linkage and current research problems. Technical report, U.S. Bureau of the Census, 1999. 3

[44] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008. 6

[45] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009. 6

[46] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *Proceedings of the 17th International Conference on World Wide Web*, pages 131–140, 2008. 6