Offline Strategies for Online Set Expansion

by

Kai Zhou

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Set expansion aims at expanding a given query seed set into a larger and more complete set by adding elements that are likely to belong to the same grouping as the elements of the query set. This thesis studies the problem of efficient set expansion; in particular, given a collection of data sets, each corresponding to an object grouping, and a query set, we develop offline strategies to preprocess and organize the data sets such that online set expansion queries can be answered efficiently. We show how those strategies can be tuned for different set expansion semantics. We also evaluate our algorithms on a real dataset, constructed from the Wikipedia tables.

Acknowledgements

First of all, I would like to thanks my supervisor Dr. Davood Rafiei, for his help and support of my research. Without his guidance and persistent support, this thesis would not have been possible. From our weekly research meetings, I have learned a lot about doing research and problem solving. I also highly appreciate his great effort in reviewing my thesis.

I would also like to express my appreciation to my fellow friends who gave me kind help when I was seeking ideas or possible solutions: Andong Wang, Jiangwei Yu, Muhammad Waqar and Yifan Wu.

Thanks to the Department of Computing Science at the University of Alberta for giving me a excellent platform to take part in advanced research in Computer Science.

Finally, the most special thanks go to my family who have loved and supported me for all my life.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Have you ever wondered about the Los Angeles Lakers roster that won NBA finals a couple of years ago but only remembered the names of a few players? Or, have you tried finding more sci-fi movies like *Interstellar* and *Inception*? We may search them on a search engine using keywords like "Los Angeles Lakers roster NBA final 2000-2003" and "sci-fi movies interstellar inception". For popular searches such as those listed, we may gather the answers after browsing a few Web pages; for less popular searches, the chance of finding an answer can be slim. These are some examples of set expansion queries, and a system supporting such queries can provide more efficient and elegant results. This is our motivation for investigating the problem of efficient set expansion.

1.1 Problem statement

Set expansion aims at expanding a small set of elements into a more complete set by searching for other elements that may also belong to the same set or grouping [He and Xin, 2011]. Here a grouping can be any collection of elements that may belong to the same set, for example put together by someone or largely agreed on by the public, and a query seed set is a subset of a larger and perhaps "more complete" set. Consider the names of all NBA teams as a set; given a small number of team names such as "Los Angeles Lakers" and "Boston Celtics" as seeds, a set expansion scheme would discover other NBA teams such as "Chicago Bulls" and "San Antonio Spurs", etc.

More formally, given a seed set: $\{x_1, x_2, ..., x_k\}$, where each $x_i \in S$, set

expansion aims at retrieving a list of other elements: $\{e_1, e_2, ..., e_n\}$, where each $e_i \in S$. Most of the time, we don't know which set is the superset S. So we need to mine the potential sets from a large collection.

It is not hard to see that a set expansion can retrieve results that may not be easy to retrieve using a search engine. In particular, when we do not know the category name or the keyword for a given seed set, it is hard to guess which query should be issued to the search engine. For example, given "Toronto Raptors", "Orlando Magic" and "Utah Jazz" as the seed set, what should be the search query? Is it "the list of all NBA teams?" Actually, "list of NBA teams that have never won a championship" is more proper than the previous one, since these three teams share this specific feature more than just being NBA teams. However, a set expansion system may be able to pickup the unifying relationships that hold for the elements in the query seed set, and return a ranked list of other elements that may share the same relationships.

Unlike an alphabetical or a chronological ordering of elements that may be found in Web pages, the result of a set expansion is usually ranked based on the similarity or the relevance to the seed set, so that the more relevant elements are ranked higher in the list.

Table 1.1 gives two sample queries and part of their expanded sets. As we can learn from the table, the top ranked elements are very relevant to the input queries. For query {*China*, *Canada*, *Australia*}, the expanded set is other country names, while for query {*The Amazing Race*, *American Idol*, *The Voice*} the expanded set is American TV shows. We also notice that the last few elements in the outputs are not that relevant to the query sets, e.g. Winnipeg and Alaska are not country names but city or state names. This is because set expansion system will rank the most relevant elements at the top while the less relevant ones are expected to be at the bottom.

We have created a lightweight web interface for some example set expansion queries. You can see more examples from there[1].

[1]https://webdocs.cs.ualberta.ca/~kzhou3/SetExpansion.php

Table 1.1: Real examples of Set Expansion input and output

input seeds	China	The Amazing Race
	Canada	American Idol
	Australia	The Voice
outputs	United States	America's Got Talent
	France	Dancing with the Stars
	Germany	Survivor
	Japan	The X Factor
	Italy	So You Think You Can Dance
	Russia	Top Chef
	Spain	America's Next Top Model
	Greece	Access Hollywood
	Great Britain	Late Night with Jimmy Fallon
	Belgium	The Price Is Right
	Netherlands	The Ellen Show
	...	...
	*Winnipeg	*Godzilla: Unleashed
	*Alaska	*Deca Sports 3
	*Sheet D	*WWE A.M. Raw

1.2 Challenges

Set expansion may be treated as a variant of a typical information retrieval task. However, unlike a typical information retrieval query, it is not sufficient in a set expansion to only retrieve a set of top ranked documents. The goal is to dig deeper and further analyze the documents or sets, and extract the most relevant terms or elements to construct a complete set.

There are a few challenges associated to set expansion; this thesis studies two specific problems. First, given a query set, how to efficiently retrieve and rank the most relevant documents or sets from a large data set? Second, given a set of relevant sets to a query set, how to merge the relevant sets or documents into a complete expanded set, which is the most meaningful expanded set for the query set? To address these challenges, we study efficient indexes and ranking strategies for set expansion queries.

1.3 Our contribution

In this thesis, we study the problem of efficient set expansion, focusing on conducting set expansion using specific local data sources without resorting to online data. As our dataset, we have extracted all the table data on Wikipedia pages, treating each column in a table as a set. This results in a dataset with millions of concept sets, which we can use in our experiments. We propose two indexing strategies for set expansion, one is based on an inverted index and another is based on asymmetric minwise hashing. We observe that each indexing strategy has its own advantages on some specific types of queries.

We develop offline strategies to preprocess and organize the data sets such that online set expansion queries can be answered efficiently. We show how those strategies can be tuned for different set expansion semantics. We also evaluate our algorithms on a real dataset, constructed from the Wikipedia tables.

1.4 Thesis overview

The rest of the thesis is organized as follows. The next chapter covers the related work. We introduce an inverted index based set expansion in Chapter 3 and a asymmetric minwise hashing based method in Chapter 4. In the context of a set expansion, we study the problem of top-k retrieval in Chapter 5. Our experimental evaluation and analysis are presented in Chapter 6. Finally we conclude this thesis in Chapter 7.

Chapter 2

Related Work

Our work relates to the lines of work on set expansion and similar set retrieval; this chapter reviews those lines of work. We will also review two indexing strategies: inverted index and locality sensitive hashing. Our proposed indexes are based on these two strategies.

2.1 Set expansion

Set expansion has received much attention lately from both industry and academia [Ghahramani and Heller, 2005, He and Xin, 2011, Wang and Cohen, 2007, Pantel et al., 2009, Sarmento et al., 2007]. Some approaches to set expansion are totally online, using search engines to retrieve relevant documents and mining similar terms and elements within documents to construct a complete set. On the other hand, corpus based approaches to set expansion assume the entire corpus is available in advance. Thus, efficient indexes can be built offline, and these indexes may be used for fast online retrieving of similar elements.

As a related work, we have to mention Google Sets[1], which used to provide an online set expansion interface; but the details of this method have not been published. Another outstanding work is the SEAL system [Wang and Cohen, 2007, 2008, 2009]. SEAL worked by fetching some web pages that contain "lists" of elements, and then aggregating and ranking these "lists" as a complete set. SEAL did an online web data extraction and processing,

[1] http://labs.google.com/sets

which is costly and does not scale up well. He and Xin studied the problem of using general-purpose web data (web lists and query logs) to expand a set of seed entities [He and Xin, 2011]. They proposed a simple yet effective quality metric to measure the expanded set, and designed two iterative thresholding algorithms to rank candidate entities. Similarly in Sarmento's work [Sarmento et al., 2007], the authors extracted the Wikipedia "list of *" pages, treating each list in a page as a concept set. They computed the co-occurrence stats of all the set elements, which can then be used for scoring during a set expansion.

The aforementioned work has largely focused on finding documents that mention a query seed set and using the HTML structure surrounding the seed set to find more elements (e.g. [He and Xin, 2011, Wang and Cohen, 2007]). This is a time-consuming process and does not scale to large number of queries or document collections. The problem studied in this thesis is efficiently supporting online set expansion queries through offline processing strategies. In particular, we assume similar sets are collected or extracted in advance, and we want to find strategies for organizing them for more efficient querying.

2.2 Similar set retrieval

A closely related line of work to set expansion is similar set retrieval [Gionis et al., 2001, Mamoulis et al., 2003], where given a set, the goal is to find other similar sets. Some commonalities between set expansion and similar set retrrieval are (1) both need to define some reasonable similarity functions between sets and (2) both approaches usually make use of indexes for retrieving those highly similar sets efficiently.

However, there are some differences between expanding a set and finding similar sets. Expanding a set is more like finding the super set of a query set, while similar sets retrieval tends to fetch other largely overlapping sets, which may have similar contents and be roughly about the same set sizes. A well-used similarity function for similar set retrieval is the Jaccard similarity [Rajaraman et al., 2012]. As a matter of fact, the Jaccard similarity function penalizes the matching sets based on their sizes; for a given fixed query set, the larger a

Table 2.1: Domain $\mathcal{D}$

set ID	elements
S_1	apple, banana, grape
S_2	apple, google, facebook

matching set is the less Jaccard similarity it will have, which is not desirable in set expansion situations where the query sets are small and the target sets are typically large (see Section 4.3 for some details).

Nevertheless, the basic goal of similar set retrieval and set expansion are very alike. Both need to retrieve relevant sets from a large collection of sets, even though the definition of relevance is a little different. There has been many work on efficiently retrieving similar sets, based on the Jaccard similarity, using an inverted index [Patil et al., 2011, Fontoura et al., 2011, Zobel and Moffat, 2006, Yan et al., 2009], locality sensitive hashing [Datar et al., 2004, Slaney and Casey, 2008], etc. Set expansion may use the relevant literature on similar set retrieval and efficiently find sets from which an answer can be constructed.

2.3 Inverted index

Inverted index is widely used in modern information retrieval systems, for example, to support keyword searches on documents [Zobel and Moffat, 2006]. An inverted index may also be used in efficiently retrieving similar sets. In this case, an inverted index consists of a directory containing all distinct elements that appear in a set and a posting list of set IDs where each element appears in. Inverted index offers a promising strategy for set expansion.

For example, consider the two sets, S_1 = {apple, banana, grape} and S_2 = {apple, google, facebook}. The data sets are shown in Table 2.1 and an inverted index is shown in Table 2.2.

Given a query set q = {apple, google}, we can first retrieve the set IDs that they appear in, which are S_1 and S_2; S_2 may be ranked higher because it contains both query elements. Having retrieved the set IDs, we can go back

Table 2.2: Inverted index

element	set IDs
apple	S_1, S_2
banana	S_1
grape	S_1
google	S_2
facebook	S_2

to each set and retrieve other set members as relevant elements to the query. In this particular case, the result set may be {apple, google, facebook} or {apple, google, facebook, banana, grape} depending on the result set size that is expected or a similarity threshold that may be desired.

As mentioned in Chapter 1, a set expansion may not retrieve all answers but only those with high relevance, hence the literature on top-k document retrieval is also relevant. In a top-k document retrieval using inverted indexes, queries are evaluated using two major approaches: *document-at-a-time* (DAAT), *term-at-a-time* (TAAT) and *Weighted-AND* (WAND) [Culpepper et al., 2012, Jonassen and Bratsberg, 2011, Patil et al., 2011, Ilyas et al., 2008]. Let us take a look into them.

2.3.1 DAAT

A DAAT approach to keyword queries simultaneously traverses the postings lists for all terms in the query [Fontoura et al., 2011]. A naive implementation of DAAT simply merges the involved postings lists and examines all the documents in the union. A min-heap is normally used to store the top-k documents during the evaluation. Whenever a new candidate document is identified, it must be scored. The computed score is then compared to the minimum score in the heap, and if it is higher, the candidate document is added to the heap. At the end of processing, the top-k documents are guaranteed to be in the heap.

There are two main factors in evaluating the performance of the DAAT algorithms: the index access cost and the scoring cost. In the case of the naive DAAT algorithm, every posting for every query term must be accessed. The

index access cost is then proportional to the sum of the sizes of the postings list for all query terms. The scoring cost includes computing the scoring function and updating the result heap.

2.3.2 TAAT

A TAAT algorithm traverses one postings list at-a-time [Fontoura et al., 2011]. The contributions from each query term to the final score of each document must be stored in an array, then be added up to the final score. The size of the array is the number of documents in the index. In the naive implementation of TAAT we must access every posting for every term. For each posting, we compute its score contribution and add it to the array.

The costs of both DAAT and TAAT algorithms are O(N), i.e. linear complexity with respect to the size of the posting list. Previous work has compared DAAT and TAAT algorithms' performance on a large TREC GOV2 document collection [Carmel and Amitay, 2006]. They found that even though DAAT and TAAT are at the same order in terms of cost, DAAT was superior than TAAT. However, unlike our work, which focuses on memory-resident indexes, the authors used the disk-base indexes for their performance evaluations.

2.3.3 WAND

In a set expansion context, we have two rounds of top-k retrievals. The first round is to retrieve top-k relevant sets. The second round is to merge the sets and produce top-k ranked relevant terms. As a result, there are two kinds of scoring functions for each round of a top-k retrieval, which can be considered as the largest part of the cost.

Many algorithms are proposed to speed up the scoring process, of which WAND [Ding and Suel, 2011, Broder et al., 2003] is a very popular one. The main intuition behind WAND is to use upper bounds on score contributions to improve query performance. For each postings list in the index, we can precompute and store the maximum score value. It is demonstrated that using the document-at-a-time approach and a two-level query evaluation method

using the WAND operator for the first stage, pruning can yield a substantial gain in efficiency at no loss in precision and recall [Broder et al., 2003].

2.4 Locality sensitive hashing

Locality sensitive hashing (LSH) may also be used to store sets and to retrieve similar sets. Unlike an inverted index which stores each set ID in the posting list of every element, LSH indexes the sets as a whole.

LSH is a widely used strategy for data clustering and nearest neighbor search [Rajaraman et al., 2012, Datar et al., 2004, Slaney and Casey, 2008, Charikar, 2002]. LSH hashes the input elements so that similar elements map to the same buckets with high probability (the number of buckets is much smaller than the universe of possible input elements). LSH differs from conventional and cryptographic hash functions because it aims to maximize the probability of a collision for similar elements [Rajaraman et al., 2012].

LSH is also a very promising way of indexing similar sets. It is based on the assumption that similar sets have higher probability to be hashed into a same bucket. Instead of hashing a set directly, we usually hash the min value signature of the set. The simplest version of the MinHash scheme uses k different hash functions, where k is a fixed integer parameter, and represents each set S by the k values of min(S) for these k functions. Those signatures represent the sets and can be hashed into corresponding buckets more efficiently.

2.4.1 Asymmetric minwise hashing

However, using MinHashes for a set expansion task is problematic [Shrivastava and Li, 2014, 2015], because its underlining Jaccard similarity has inherent bias towards smaller sets.

For example, consider these two sets, $S_1 = \{\text{apple, banana, grape}\}$, $S_2 = \{\text{apple, google, facebook, microsoft, linkedin, amazon, intel, ibm, dropbox}\}$. If the query seed set S_q is {apple, google}, then the Jaccard similarity between S_q and S_1 is $J(S_q, S_1) = \frac{1}{4} = 0.25$, and that between S_q and S_2 is $J(S_q, S_2) = \frac{2}{9} = 0.22$. As a result, simply based on the Jaccard similarity, S_1, having higher

Jaccard similarity, is more similar to S_q than S_2, which however should not be correct in this case, since S_q and S_2 are both sets of technology companies in the US while S_1 is just a set of fruits.

The issue here is that the Jaccard similarity penalizes larger sets, or it favors small sets, and this can hurt the set expansion whose goal is to find larger and more complete sets. In order to utilize MinHash as a tool for set expansion under the overlap similarity, we have to do some modifications, such as adding some asymmetric factors into the similarity function to alleviate the large set size penalty, see Chapter 4.3 for details.

In our work, we propose and experiment with two indexing strategies for set expansion: inverted index and asymmetric minwise hashing. The methods are presented in Chapter 3 and 4 and a preliminary evaluation of their performance is presented in Chapter 6.

Chapter 3

Set Expansion using Inverted Index

Inverted index can work as a key component for retrieving relevant sets during a set expansion process. In this chapter, we study the problem of efficient set expansion using an inverted index.

3.1 Inverted index

Let $\mathcal{S}$ represent a set and $\mathcal{E}$ be the element in the set. In an inverted index, each element $\mathcal{E} \in \mathcal{S}$ is associated with a posting list that contains set IDs of all the sets that contain element $\mathcal{E}$. If we donate a posting list with $\mathcal{P}$, our dataset consists a collection of sets like $\mathcal{S}_i = \{\mathcal{E}_1, \mathcal{E}_2, ..., \mathcal{E}_n\}$ and a set of posting lists $\mathcal{P}_{\mathcal{E}_i} = \{\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_n\}$, extracted from the sets. Given a set expansion query $\mathcal{Q} = \{\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3\}$, we retrieve all the posting lists $(\mathcal{P}_{\mathcal{E}_1}, \mathcal{P}_{\mathcal{E}_2}, \mathcal{P}_{\mathcal{E}_3})$ using the inverted index, which will be intersected or merged to produce a final result.

As described in Chapter 2.3, with an inverted index constructed on all elements, the first step for expanding a query set is to retrieve the posting list $\mathcal{P}$ of each element in the query set. Different semantics may be used to merge the posting lists $\mathcal{P}$; this results in a set of set IDs, which can be used to fetch more relevant elements. Lastly, we merge those elements based on some scoring function to get a ranked list of elements, referred to as an expanded set. Algorithm 1 is a general inverted index based algorithm for set expansion.

The ***merge*** function in steps 5 and 9, and the ***rank*** function in step 10

Algorithm 1 Set Expansion Using Inverted Index

Input: Query set: Q_set={element1, element2, ..., elementM}
Output: Expanded set: E_set={element1, element2, ..., elementN}, N $\geq$ M
Require: Dataset: $\mathcal{D}$={set:elements}; index: $\mathcal{D}^{Inverted}$ = {element:sets}.
Steps:

Initialize posting list $\mathcal{P} = \{\}$, expanded set $E\_set = \{\}$
Retrieve all relevant set IDs into $\mathcal{P}$:
    for e **in** Q_set:
        $p \leftarrow \mathcal{D}^{Inverted}$.get($e$)
        $\mathcal{P} \leftarrow$ ***merge***(p, $\mathcal{P}$)
Retrieve all the relevant elements into E_set:
    for s **in** $\mathcal{P}$:
        elements $\leftarrow \mathcal{D}$.get(s)
        $E\_set \leftarrow$ ***merge***(elements, E_set)
rank(E_set)
Return E_set

can implement different query semantics, such as merge by union, intersection or frequency count, and use different algorithms such as quick sort or heap sort. More details are discussed in Chapter 3.2.

3.1.1 Algorithm cost analysis

The computation cost can be estimated based on the query set size and some statistics of the dataset. Let n denote the query set size and p_i denote the size of a posting list i. Let the average set size be μ. The cost includes two parts: (1) the cost of getting all the relevant sets and (2) the cost of merging and ranking all the relevant elements. For the first part, we need to aggregate the posting lists of the query set, and its cost is the summation of the posting lists $P = \sum_{i=1}^{n} p_i$. For the second part, the cost varies for different merge and rank methods. If we denote the two costs respectively by C_{merge} and C_{rank}, the total cost becomes

$$C = O(P) + C_{merge} + C_{rank}. \tag{3.1}$$

3.2 Merge and rank methods

An inverted index on set elements can help us retrieve all relevant set IDs, but we need to find some effective ways to merge those sets and produce meaningful ranked list as an expanded result. Some naive ways to merge is to take the union or intersection of the sets. When we take the union, the expanded result may be too large and contain many irrelevant or noisy elements. On the other hand, when taking intersection of the sets, it may not always guarantee that the intersection of all the sets is not empty.

Let us take the two sets used in Chapter 2.4.1 for an example. The two relevant sets for the query set S_q = {apple, google} are S_1 = {apple, banana, grape} and S_2 = {apple, google, facebook, microsoft, linkedin, amazon, intel, ibm, dropbox}. If we take the union of S_1 and S_2 as the expanded set, then we can get E_set = {apple, banana, grape, google, facebook, microsoft, linkedin, amazon, intel, ibm, dropbox}. This expanded set is not only too large, but also self-inconsistent as a meaningful set. On the contrary, when we take the intersection of S_1 and S_2 as the expanded set E_set = {apple}, even though it is not an empty set, it is obviously not an expanded set for the query set.

As a result, we need more sophisticated strategies to merge the sets and rank the elements. We have investigated three ranking strategies.

3.2.1 Rank by Overlap Similarity

Let us present the formal definition of overlap similarity first. For sets S_1 and S_2, the overlap similarity between S_1 and S_2 is the ratio of the number of elements of their intersection and the size of S_1:

$$O(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1|}.$$

One way to rank the query result is to first rank the sets from which the result is derived. Sets may be ranked based on their overlap similarities with the query set. The sets with higher overlap similarities are ranked higher in the list. To construct an expanded set, we can scan the list from the top and fetch the elements of those sets to build the expanded set. That means the

elements of the first set in the list are all ranked higher than the elements of the second set, and the elements of the second set are ranked higher than those of the third, etc.

For example, consider the query {*Canada*, *US*} with the matching sets S_1={*Canada*, *US*, *China*, *Noise1*}, S_2={*Canada*, *Australia*, *Noise2*} and S_3 {*US*, *Australia*, *Noise3*}. S_1 has an overlap similarity of 2/2, while both S_2 and S_3 have an overlap similarity of 1/2. All the elements in S_1 (including *Noise1*) are ranked higher than elements in S_2 and S_3. This method is called *Rank by Overlap Similarity* (ROS).

Let us analyze the cost of this ranking method. First, it needs to sort the sets by their overlap similarity with the query set. We know that the size of the posting list is P, so the cost for sorting is $C_{sort} = O(P * logP)$. Then we can merge the elements from this ranked list of sets. Suppose X is a random variable representing the size of each set; then the size of the merged list $C_{merge} = O(\sum_{i=1}^{P} X_i)$. Since X_i is independent and identically distributed (i.i.d.), we have

$$\begin{aligned} E(\sum_{i=1}^{P} X_i) &= \sum_{i=1}^{P} E(X_i) \\ &= PE(X) \\ &= P\mu \end{aligned}$$

So the the cost for set expansion rank by overlap similarity is

$$\begin{aligned} C &= O(P) + C_{merge} + C_{rank} \\ &= O(P) + O(P\mu) + O(P * logP) \\ &= O(P + P\mu + P * logP) \end{aligned}$$

.

3.2.2 Rank by Frequency Count

The method described above can be a bit rough sometimes, in that, it only considers the similarity between the sets. Not all elements in a set may be

relevant to the query set, and some elements may be more relevant than others. In the last example, we notice that element *Noise1* is also ranked higher than any other elements in S_2 and S_3, such as *Australia*. In this case, we expect an extended set to be a set of country names, and clearly Noise1 is not what we expect in the result. Our ranking by *Frequency Count* (FC) takes the element similarity into consideration to get more reasonable results.

The basic assumption for our ranking here is that if an element co-occur with another element in many sets, these two elements are likely to be related. For example, *Canada* and *US* typically co-occur in many sets, such as the set of all countries, the set of all North American countries or the set of largest countries in the world. In this case, many set ids in the posting list of Canada are expected to appear in the posting list of USA as well, and vice versa. Based on this observation, we assume that the more frequent two elements appear in the same sets, the more relevant those two elements are.

Furthermore, we can combine the frequency count and overlap similarity rank together to produce a potentially more meaningful ranked list of elements. To be specific, if a set has a higher overlap similarity with the query set, the elements in this set should be more relevant with the query set. Taking this into consideration, we can assign the set overlap similarity as a weight to each set when doing the counting of element's frequency.

For example, consider again the query set {*Canada*, *US*} and the matching sets S_1={*Canada*, *US*, *China*, *Noise1*}, S_2={*Canada*, *Australia*, *Noise2*} and S_3={*US*, *Australia*, *Noise3*}. If we only count the frequency of elements, *China* and *Noise2* and *Noise3* will have the same frequency. However, the overlap similarity between the query and S_1 is 2/2= 1 and that between the query and S_2 and S_3 is 1/2. If the sets S_1, S_2 and S_3 are weighted based on their overlap similarities, *China* will gain a weight of w(S_1)=1 whereas, *Noise2* only appears in S_2 and will have the weight of w(S_2)=0.5, *Noise3* also only has the weight of w(S_3)=0.5. As a result, *China* is ranked higher than *Noise2* and *Noise3* in the expanded set.

In order to accumulate the frequency of each element in all the posting lists, we can store the elements in a hash table, where the key is the element

and the value is its frequency. Using this extra space, we can obtain all element frequencies in just one pass of the posting lists. Having the element frequencies in a hash table, we can sort the elements by their frequencies so that the most relevant elements are ranked on the top of the list.

Algorithm 2 Frequency Count

Input: Query set: Q_set={element1, element2, ..., elementM}
Output: Expanded set: E_set={element1, element2, ..., elementN}, N $\geq$ M
Require: Dataset: $\mathcal{D}$={set:elements}; index: $\mathcal{D}^{Inverted}$ = {element:sets}.
Steps:

```
1: Initialize posting list P={} as a hash table to count the frequency of sets
2: Retrieve all relevant set IDs into P:
3:     for e in Q_set:
4:         p ← D^Inverted.get(e)
5:         for s in p:
6:             if s in P:
7:                 P[s]++
8:             else:
9:                 P[s] = 1
10: Initialize E_set={} as a hash table to count the frequency of elements
11: Retrieve all the relevant elements into E_set:
12:     for s in P:
13:         elements ← D.get(s)
14:         weight = P[s]
15:         for e in elements:
16:             if e in E_set:
17:                 E_set[e] += weight
18:             else:
19:                 E_set[e] = weight
20: sort(E_set) by the weights
21: Return E_set
```

Algorithm 2 is the complete algorithm of Frequency Count. It contains two major phases. In the first phase (Steps 3 to 9), we merge all the posting lists so that we know the similarity weights between the query set Q_set and other relevant sets in $\mathcal{D}$, which is

$$\mathcal{P}(Q\_set) = \{(s, w) | s \in \mathcal{D} \wedge w = |s \cap Q\_set|\}.$$

In the second phase (Steps 12 to 19), we use the aggregated posting list and

elements' frequency counts to produce the merged elements list, which is

$$E\_set = \{(e, \sum w_i) | (s, w_i) \in \mathcal{P}(Q\_set) \wedge e \in s\}.$$

Let us analyze the cost of this ranking method. The parts on retrieving the relevant sets and merging are very similar to the previous method. What is different is the sorting part. In the frequency count method, we need to sort the elements based on their frequency counts instead of their overlap similarities. So the cost for sorting becomes $C_{rank} = O(P\mu * logP\mu)$ and total cost for set expansion in Rank by Frequency Count is

$$\begin{aligned} C &= O(P) + C_{merge} + C_{rank} && (3.2)\\ &= O(P) + O(P\mu) + O(P\mu * logP\mu) && (3.3)\\ &= O(P + P\mu + P\mu * logP\mu). && (3.4) \end{aligned}$$

3.2.3 Rank by Frequency and Inverse Frequency Count

Inspired by the two methods discussed above and the widely used Term Frequency-Inverse Document Frequency (TF-IDF) weighting in information retrieval and text mining tasks [Rajaraman et al., 2012, Wu et al., 2008, Salton and Buckley, 1988], we can develop another ranking method for elements in an expanded set, which is called *Frequency and Inverse Frequency Count* (F-IFC).

TF-IDF weighting is a statistical measure used to evaluate how important a word is to a document in a collection or corpus [Wu et al., 2008]. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. Variations of the TF-IDF weighting scheme are often used by search engines as a way of scoring and ranking a document relevance to a given user query.

As TF in our case, we consider weighting the elements of a list based on the number of times an element appears. The TF weighting scheme may be applied in the context of set expansion by treating a set as a document and an element of a set as a term of the document. However, there are some differences between a set and a document. Since an element can appear at most once in a set, the TF is either 1 or 0. If we treat TF for a document as a

probability distribution over terms, then the probability that a term e appears in set S can be expressed as:

$$TF(e,\ S) = \begin{cases} \frac{1}{|S|}, e \in S \\ \frac{0}{|S|}, e \notin S \end{cases} \tag{3.5}$$

Again in an information retrieval context the Inverse Document Frequency (IDF) measures how important a term is. In computing a TF score, all terms are considered equally important. However it is known that certain terms, such as "is", "of", and "that", may appear many times and in a large number of documents with little importance. The IDF weighting weighs down the frequent terms while scales up the rare ones.

In a set expansion context, let N denote the number of sets and N_e be the number of sets with element e in them. Then the IDF score of e is

$$IDF(e) = log(N/N_e). \tag{3.6}$$

Let us look at an example here. Consider a set containing 100 elements where the element *cat* is in it. The term frequency (i.e., TF) for cat is then (1/100) = 0.01. Now, assume we have one thousand sets and the element *cat* appears in ten of these. Then, the inverse document frequency (i.e., IDF) is calculated as log(1,000 / 10) = 2. Thus, the TF-IDF weight of that element is the product of these quantities: 0.01 * 2 = 0.02.

The TF-IDF weighting schema can also be combined with set overlap weight as in the Frequency Count method. Algorithm 3 is the complete algorithm of Frequency and Inverse Frequency Count.

The TF-IDF function in the algorithm is using Eq. 3.5 and 3.6 to get the tf-idf score. The cost of this algorithm is quite similar to that of FC, since the only difference is the way the weights are calculated.

3.2.4 A comparisons of different merge and rank methods

We have developed a few merge and rank methods, as discussed above, each with its own advantages under different settings.

Algorithm 3 Frequency and Inverse Frequency Count

Input: Query set: Q_set={element1, element2, ..., elementM}
Output: Expanded set: E_set={element1, element2, ..., elementN}, N $\geq$ M
Require: Dataset: $\mathcal{D}$={set:elements}; index: $\mathcal{D}^{Inverted}$ = {element:sets}.
Steps:

```
Initialize posting list P={} as a hash table to count the frequency of sets
Retrieve all relevant set IDs into P:
    for e in Q_set:
        p ← D^Inverted.get(e)
        for s in p:
            if s in P:
                P[s]++
            else:
                P[s] = 1
Initialize E_set={} as a hash table to count the frequency of elements
Retrieve all the relevant elements into E_set:
    for s in P:
        elements ← D.get(s)
        weight = P[s]
        for e in elements:
            if e in E_set:
                E_set[e] += weight*TF-IDF(e, s)
            else:
                E_set[e] = weight*TF-IDF(e, s)
sort(E_set) by the weights
Return E_set
```

For example, if we want to expand a query set as large as possible, merging by union operation is the best choice. On the other hand, if we only need to find the most relevant elements, merging by intersection suits our goal. Rank by overlap similarity works the best when the dataset does not contain much noise. Rank by frequency and inversed frequency count are two similar ways of noise-tolerant scoring strategies. Frequency count takes the co-occurrence statistics into consideration, while inversed frequency count brings some regularization penalties to those too frequent stop-words. We will do an experimental evaluation of these ranking methods to see if one strategy works better in general set expansion cases.

3.3 Some issues with inverted index based set expansion

Based on the cost analysis, we know that the inverted index based methods cannot scale well when the posting list sizes increase. As shown in Eq. 3.4, it takes linearithmic time (NlogN) to sort the list of elements. Also, we have found that the timing performance of them suffers when the distribution of the elements is skewed, meaning some elements appear far more frequently in different sets, resulting in long posting lists. This is also consistent with earlier findings [Chaudhuri et al., 2007] and leads us to finding other strategies, such as locality sensitive hashing, to be discussed next.

Chapter 4

Set Expansion using Hashing Mechanism

In this chapter, we will discuss how we can use the locality sensitive hashing mechanism and its modification to solve set expansion problems.

4.1 MinHash signatures

First of all, we know that hashing a set directly is very costly. So we find a way to use MinHash signatures to represent the original sets.

Given a set $S \subseteq \mathcal{D} = \{1, 2, ..., N\}$ and a permutation $\pi : \mathcal{D} \to \mathcal{D}'$, the minwise hashing of S with respect to π, denoted as $h_\pi(S)$, is the element in S with the least index in the permutation. In general, $\mathcal{D}$ may not be a set of integers, but any such set can be mapped to a set of integers.

A commonly used similarity function between sets is the Jaccard similarity, which is defined for sets S_1 and S_2 as the ratio of the number of elements of their intersection and the number of elements of their union:

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}.$$

Given the sets S_1 and S_2, it has been shown that the probability that these two sets have the same MinHash is their Jaccard similarity $J(S_1, S_2)$ [Broder, 1997],

$$Pr_\pi(h_\pi(S_1) = h_\pi(S_2)) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = J(S_1, S_2).$$

Having this property, we can use MinHash signatures to represent the sets,

so that sets with high Jaccard similarity will have a high probability of sharing an identical MinHash signature.

4.2 Locality sensitive hashing

Having the MinHash signatures of all the sets, we want to place the most similar sets in near locations. One general approach to LSH is to hash items several times, in such a way that similar items are more likely to be hashed to the same bucket than dissimilar items are.

We implement LSH functions using the banding technique [Rajaraman et al., 2012]. To be specific, we divide the MinHash signature into b bands with r values in each band. We take each mini-band as a MinHash signature and hash it into a corresponding bucket. Thus we hash the whole MinHash signature b times, and this is expected to increase the collision probabilities for similar sets. Suppose the Jaccard similarity between two sets is s, then the probability that the two sets will be hashed into a same bucket (become a candidate pair) is

$$1 - (1 - s^r)^b.$$

Please refer to Chapter 3.4 in Rajaraman's book [Rajaraman et al., 2012] for detailed derivation.

This probability function has the form of an S-curve, as shown in Figure 4.1. By adjusting parameters b and r, we can set the *threshold*, which is the steepest rise point in the curve, to a similarity value s that we desire.

4.3 Asymmetric minwise hashing

As our example in Chapter 2.4.1 shows, the use of MinHash for set expansion is problematic since the Jaccard similarity biases towards smaller sets. Recently Shrivastava and Li [Shrivastava and Li, 2015] present a simple asymmetric trick to fix this problem and to make the overall probability of matching the MinHash monotonic with overlap similarity.

The basic idea is that we reduce the advantage of small sets by adding

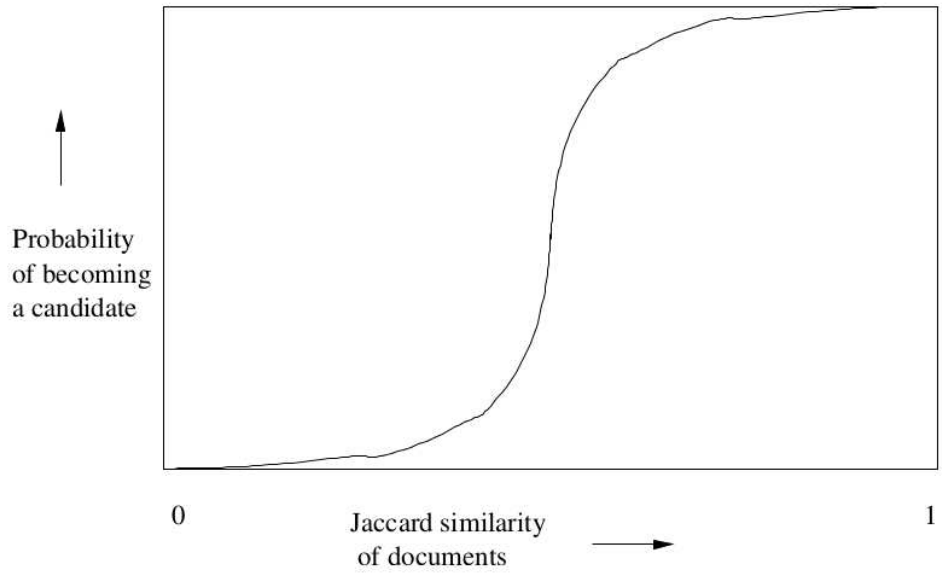

Figure 4.1: LSH's collision probability S-curve

some random irrelevant elements. Now, we will describe in detail how do add this asymmetry into a traditional MinHash. In order to utilize MinHash as a tool for set expansion under the overlap similarity, we have to do some modifications.

Suppose sets are represented using binary vectors. We define a constant M as

$$M = \max_{S \in \mathcal{D}} |S|$$

where M is the maximum set size in the collection. Then, we apply the following transformations $P'()$ and $Q'()$ respectively to data and query sets.

$$P'(x) = [x; 1; 1; 1; ...; 1; 0; 0; ...; 0]$$
$$Q'(q) = [q; 0; 0; 0; ...; 0; 0; 0; ...; 0]$$

$P'(x)$ appends $M - f_x$ 1s and f_x zeros to x, where f_x is the size of set x. $Q'(q)$ appends M zeros to q. Now we can get the Jaccard similarity between $P'(x)$ and $Q'(q)$:

$$J(P', Q') = \frac{|P' \cap Q'|}{|P' \cup Q'|} = \frac{a}{f_x + f_q - a + M - f_x} = \frac{a}{M + f_q - a}$$

where f_q is the size of query set q, a is the overlap between the two sets. We know that the overlap similarity between x and q is $O(x, q) = \frac{a}{f_q}$. As we can see

$$J(P', Q') = \frac{a}{M + f_q - a}.$$

For a fixed f_q and large constant M, $J(P', Q')$ increases monotonically with a, which means ranking the results based on $J(P', Q')$ would provide a reasonably good estimation of $O(x, q) = \frac{a}{f_q}$. In our dataset, the constant M is very large, close to four thousand. So this asymmetric transformation can be taken as a bridge between Jaccard similarity and the overlap similarity in the MinHash framework.

The asymmetric MinHash in Shrivastava and Li's work [Shrivastava and Li, 2015] is presented for binary vectors but because of the sparsity of those vectors, a binary representation leads to large storage overhead and less efficient access. As a solution, we treat each set as a set of integers (or longs) and modify the transformations as follows:

$$P'(x) = x \cup \{(M - f_x) \textit{ random integers beyond the sets}\}, \tag{4.1}$$

$$Q'(q) = q. \tag{4.2}$$

All the results presented for binary vectors can be extended to the new representations. We refer to this method as MH-ALSH.

4.4 Set expansion using MH-ALSH

Now we can use the MH-ALSH strategy to do the set expansion. Based on previous explanations, we know that sets that have a large overlap similarity are more likely to be hashed into same index buckets using MH-ALSH. So we can use those indexes to retrieve similar sets, which is extremely useful when we do the set expansion.

To do so, the first step is to get the MinHash signature of the query set. Then, use the MH-ALSH hashing mechanism to hash it into some buckets and retrieve the other sets in those buckets that are the relevant to the query set. After the similar sets are retrieved, we can use the merge and rank methods

(discussed in Chapter 3.2 to do a set expansion. The complete algorithm is in Algorithm 4.

Algorithm 4 Set Expansion Using MH-ALSH Index

Input: Query set: Q_set={element1, element2, ..., elementM}
Output: Expanded set: E_set={element1, element2, ..., elementN}, N $\geq$ M
Require: Dataset: $\mathcal{D}$={set:elements}; index: $\mathcal{D}^{MH-ALSH}$ = {bucket:sets}.
Steps:

```
Calculate indexes for the query set:
    Sig_Q ← MinHash signature of the query set Q_set
    bucket_IDs ← locality hashing Sig_Q to different buckets
Initialize posting list P = {}, expanded set E_set = {}
Retrieve all relevant set IDs into P:
    for b in bucket_IDs:
        sets ← D^{MH-ALSH}.get(b)
        P ← merge(sets, P)
Retrieve all the relevant elements into E_set:
    for s in P:
        elements ← D.get(s)
        E_set ← merge(elements, E_set)
sort(E_set)
Return E_set
```

The cost of a MH-ALSH based set expansion is similar to that of an inverted index. Let us take the cost of rank by frequency count as an example. The cost is

$$C = O(P + P\mu + P\mu * logP\mu).$$

However, the posting list size P here is different. It is not dataset dependent anymore, which means we can control the range of P by tuning the hashing parameters. As discussed in Section 3.3, an inverted index based set expansion suffers when the distribution of the elements is skewed which leads to long posting lists. In a MH-ALSH based set expansion, we can tune the hashing parameters (such as b and r) to adjust the collision similarity threshold, only allowing sets with high similarity to be hashed in a same bucket, so that each bucket will not have too many sets to cause the long posting lists problem. With a controlled size of P, an MH-ALSH based set expansion is expected to have a faster query response time than an inverted index based approach.

The fundamental difference between the inverted index based method and

MH-ALSH based method is that inverted index returns all the overlapped sets, including sets that have only one element in common with the query set, while the similarity threshold for set collisions can be controlled in MH-ALSH.

Chapter 5

Top-k Retrieval

Sometimes we want to fix or bound the size of an expanded set. In particular, given a ranking of the results, we may want to only return the top k results. A naive way of implementing a top-k retrieval is to do a full set expansion first and to return only the first k results. For a full set expansion, our algorithms from the previous chapters can be used. This is clearly a waste of computing resources since we are computing results which may not be used. An alternative method is to do a set expansion while keeping the size of the expanded result not surpass k. The second method is expected to be more efficient (both in time and space costs) than the naive one since it does not need to build a complete expended set beforehand. This chapter focuses on efficient top-k set expansion.

There are a number of top-k algorithms in the literature (e.g. [Ilyas et al., 2008, Yan et al., 2009]) that can be used. A major data structure being used is a priority queue (or a min-heap), which is a very efficient data structure for maintaining a ranked list dynamically [Knuth, 1968]. The advantage of using a fixed size priority queue is that it can constrain the cost of retrieving top-k under a k factor.

To be specific, if we are sorting the whole list to get the top-k elements, the cost would be in the scale of O($nlogn$), where n is the length of the list. However, when using a priority queue, we only need to do a linear scan of the list and maintain the top-k candidates in the priority queue. The cost for maintaining a priority queue is O($logk$). As a result, the total cost is reduced

from O($nlogn$) to O($nlogk$). Most of the time, k is much less than n. So we have reduced the cost a lot.

Algorithm 5 describes how we can use a priority queue to build the top-k results without constructing a complete expended set. The first phase (Steps 1 to 9) is still to retrieve the relevant set IDs from the posting lists as the same with the set expansion algorithms. The second phase (Steps 10 to 19) is to fetch and accumulate set elements from those set IDs. After we get the elements, we can use a fixed size priority queue to collect the top k elements for the expanded set. The cost of this algorithm becomes:

$$C = O(P + P\mu + P\mu * logk).$$

Algorithm 5 Top-k retrieving

Input: Query set: *Q_set*={element1, element2, ..., elementM}
Output: Top-k expanded set: *Top_set*={element1, element2, ..., elementK}
Require: Dataset: $\mathcal{D}$={set:elements}; index: $\mathcal{D}^{Inverted}$ = {element:sets}.
Steps:

Initialize posting list $\mathcal{P}$={} as a hash table to count the frequency of sets
Retrieve all relevant set IDs into $\mathcal{P}$:
    for e **in** Q_set:
        $p \leftarrow \mathcal{D}^{Inverted}$.get($e$)
        for s **in** p:
            if s **in** $\mathcal{P}$:
                $\mathcal{P}[s]$++
            else:
                $\mathcal{P}[s] = 1$
Initialize *E_set*={} as a hash table to count the frequency of elements
Expand the set into a larger set size of k:
    for s **in** $\mathcal{P}$:
        elements $\leftarrow \mathcal{D}$.get(s)
        weight = $\mathcal{P}[s]$
        for e **in** elements:
            if e **in** *E_set*:
                E_set[e] += weight
            else:
                E_set[e] = weight
Initialize a priority queue PQ size of k to store the top-k elements
    for *key*, *value* in *E_set*:
        put *key*, *value* into PQ
    Top_set $\leftarrow$ elements that remain in the PQ
Return *Top_set*

Chapter 6

Experiments and Discussions

In this chapter, we report our experimental evaluations of the performance of our algorithms, including the running time and the accuracy of the results.

6.1 Dataset

For our experiments in this thesis, we use a set of lists collected from the Web. More specifically, we extract all tables from Wikipedia and treat each column of a table as a set based on the observation that each column has a domain and the values are drawn from that domain or set; see Figure 6.1 for an example of table on the Wikipedia website. In this table, we can extract sets like {Steve Nash, Kobe Bryant, Tracy McGrady, Tim Duncan, Yao Ming} and {Phoenix Suns, Los Angeles Lakers, Houston Rockets, San Antonio Spurs, Houston Rockets} etc.

Our preliminary experiments reveal that these sets are highly useful for set expansion tasks; for example, the sets are very diverse and have rich information that cover most domains of interest. On the other hand, the data is inherently noisy and we need to do some filtering to clean it.

In particular, we remove any set that has less than three distinct elements and sets that consist of all numbers, since those small and numerical sets is not very useful for set expansion tasks. We remove duplicates in each column to reduce a list into a pure set. We also delete a few frequent keywords such as *unknown, tba, total*. After those cleanings, we obtain 1,707,913 sets, with the average set size 11, minimum size 3, and maximum size 3,823. The standard

Western Conference All-Stars

Pos.	Player	Team	# of Selections
Starters			
PG	Steve Nash	Phoenix Suns	4th
SG	Kobe Bryant	Los Angeles Lakers	8th
SF	Tracy McGrady	Houston Rockets	6th
PF	Tim Duncan	San Antonio Spurs	8th
C	Yao Ming	Houston Rockets	4th
Reserves			
SG	Ray Allen	Seattle SuperSonics	6th
PF	Elton Brand	Los Angeles Clippers	2nd
PF	Kevin Garnett	Minnesota Timberwolves	9th
PF/C	Pau Gasol	Memphis Grizzlies	1st
SF	Shawn Marion	Phoenix Suns	3rd
PF	Dirk Nowitzki	Dallas Mavericks	5th
PG	Tony Parker	San Antonio Spurs	1st

Figure 6.1: A sample table on the Wikipedia

deviation of the set size is 23. We invert the sets, obtaining for each element a list of set IDs where the element is listed in. As for the posting list statistics, the average posting list size is 3, minimum is 1, maximum is 27,959 and the standard deviation of the size is 46.

6.2 Experiment settings

We ran all the experiments on a Linux machine with 16 AMD CPU cores, 2300 MHz each core, and 96GB RAM, running Ubuntu 12.04 LTS. The methods were all implemented in Python 2.7. We evaluate the different schemes on the

actual task of retrieving top-ranked elements.

A difficulty in evaluating the accuracy of a set expansion algorithm is the lack of a publicly available dataset containing query seed sets and ground truth of the "concept sets". Therefore, we construct our query sets and their matching data sets as follows:

First, we get the stats about the posting list size of each element. Then, we randomly pick elements with the posting list size in the range 3 to 10,000 with the goal of covering the whole spectrum. At last, for each seed element, we fetch more relevant elements that co-occur in the same set with the seed element (note that a seed element may appear in many sets, but we only pick one set as the source set). Those correlated elements build up our query set. The original set that those elements are derived from is the ground truth of the expanded set. For example, if we pick "swimming" as a seed element and its posting list size is around 500. We then randomly pick some other elements from those sets where "swimming" appears in and add them to the query set, such as "running" and "cycling". Then the query set is {"swimming", "running", "cycling"}. We also vary the size of each query seed set from 3 to 20.

6.3 Results

In this section, we present the timing and accuracy performance results of each set expansion algorithm.

6.3.1 Timing performances for set expansion

As our work focus on offline strategies for online set expansion, one of our main goals is to make sure online search can response as fast as possible. So we have done extensive experiments about timing.

From the algorithm complexity analysis in Chapter 3.1.1, we know that posting list size is the major factor that influence the query response time. Thus we create an experimental query set by varying the set's total posting list sizes. There are one thousand query sets, with each set's total posting list

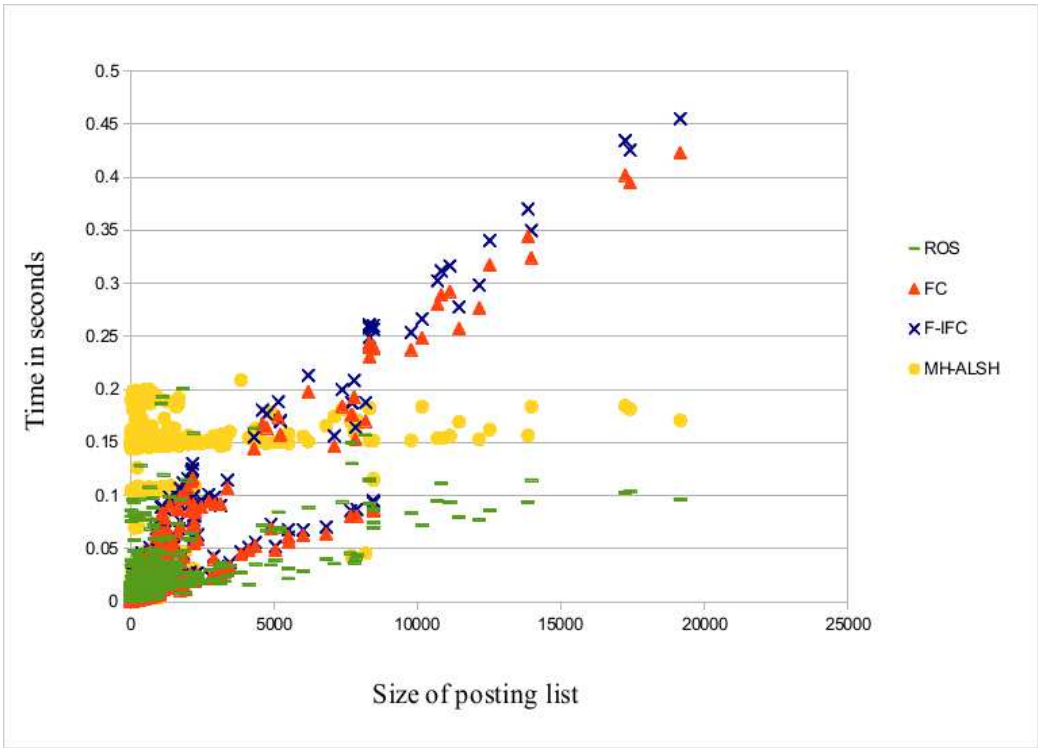

Figure 6.2: Timing of query response for set expansion

size ranging from 3 to 10,000. The reason why we choose the sum of all the posting list sizes is that it represents the worst case possible when merging the posting lists. We run the experiments 10 times and record the mean value of all the running time. The result is shown in Figure 6.2.

We can see that MH-ALSH can nearly maintain a constant response time for different posting list sizes (MH-ALSH's timing performances of different merge and rank methods are very close, so we only present the result of MH-ALSH using Rank by Frequency Count).

On the other hand, inverted index based methods using Rank by Overlap Similarity (ROS), Frequency Count (FC) and Frequency-Inverted Frequency Count (F-IFC), have very different timing performances. The naive one, ROS, has the best timing performance, since it is quite easy and straightforward, does not need much computation in the **merge** and **rank** steps. But later we will see that the accuracy performance of ROS is not as good as the others. Both FC and F-IFC have the long posting list problem, taking much time to process when the posting list is large. This confirms the cost analysis for FC

and F-IFC, which is $O(P + P\mu + P\mu * logP\mu)$. The computation time grows linearithmically with the posting list size of the query times the average set size μ of the dataset. One thing to mention is that, the worst running time performance in this experiment setting is about 0.45 seconds, which is just 0.25 seconds above the average performance. Although it is not a big absolute difference, it represents the scalability of the algorithms. When we further increase the size of the dataset, the gap will become more obvious.

We can see that all the queries are processed within one second, even though the dataset contains more than one million sets. This is because of the efficient indexes we have built. While other online set expansion systems, such as SEAL [Wang and Cohen, 2007], typically take $10 \sim 20$ seconds for one single query. Since they need to retrieve relevant documents first, then fetch elements from the documents on the fly. This response time difference is the biggest advantage for the corpus based offline strategies over the Web based online strategies.

6.3.2 Timing performances for top-k retrieval

The experimental results for top-k are shown in Figure 6.3. As a baseline for comparison, we have also plotted the case where all relevant results (and not just top k) is returned. We can see that if k is set to a small value, the processing time is very fast. This will not happen if we use the first kind of top-k retrieval method discussed in Chapter 5, which needs to construct the complete ranked expended set first.

6.3.3 Accuracy performances for merge and rank methods

In order to evaluate the accuracy performance of different merge and rank methods, we measure the top-100 precision and recall using the ground truth sets we have created in Chapter 6.2. The result is presented in Figure 6.4. We can see that both FC and F-IFC are much better than the naive ranking method. This result confirms our hypothesis in Chapter 3.2.2, that taking the element's co-occurrence statistics into consideration tends to get more accurate

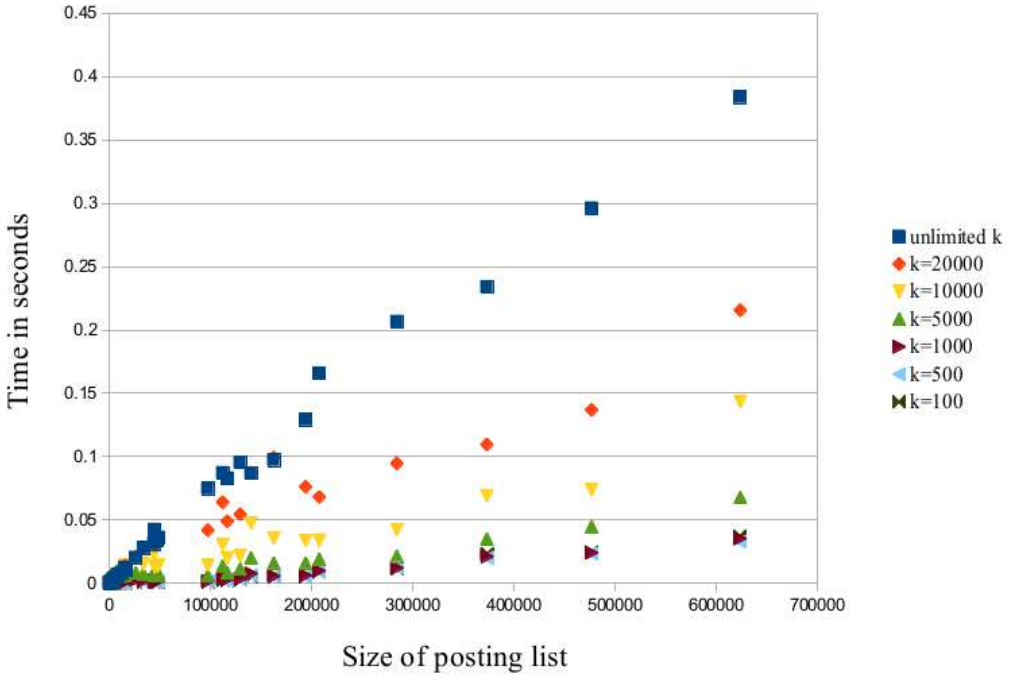

Figure 6.3: Timing for different k in top-k retrieval

results.

We can also find that the performance of FC is a little better than the F-IFC's. This may be that the term frequency and inversed document frequency weight is more appropriate in the documents context, where each term can appear multiple times in the documents, so the TF and IDF weights are more representative of the terms. While in the context of sets, the term TF is less meaningful because an element can exist at most once in a set and the maximum frequency of an element in a set is fixed at 1/(size of the set).

6.3.4 MH-ALSH v.s. LSH

In LSH functions families, the main parameters are the number of hashing functions for the pseudo permutations and how we divide a signature into b bands. By varying the number of hashing functions and the banding parameters b and r, we can adjust the similarity threshold in the S-curve of the collision probability [Leskovec et al., 2014]. To be specific, we vary the number

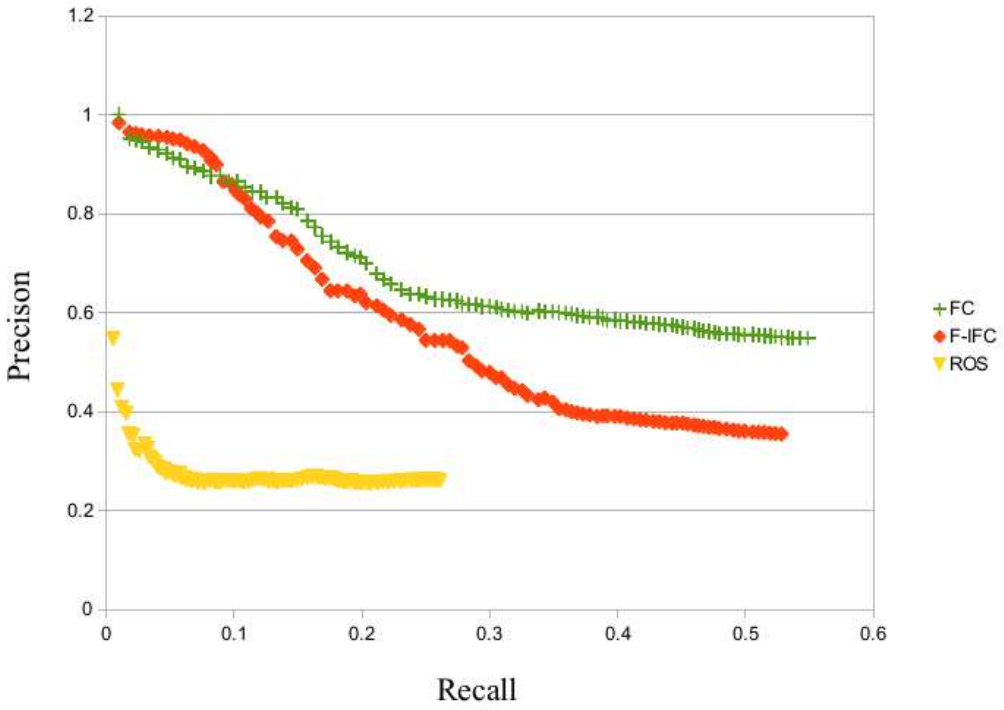

Figure 6.4: Top-100 retrieving precision and recall for different merge and rank methods. Higher precision at a given recall is better

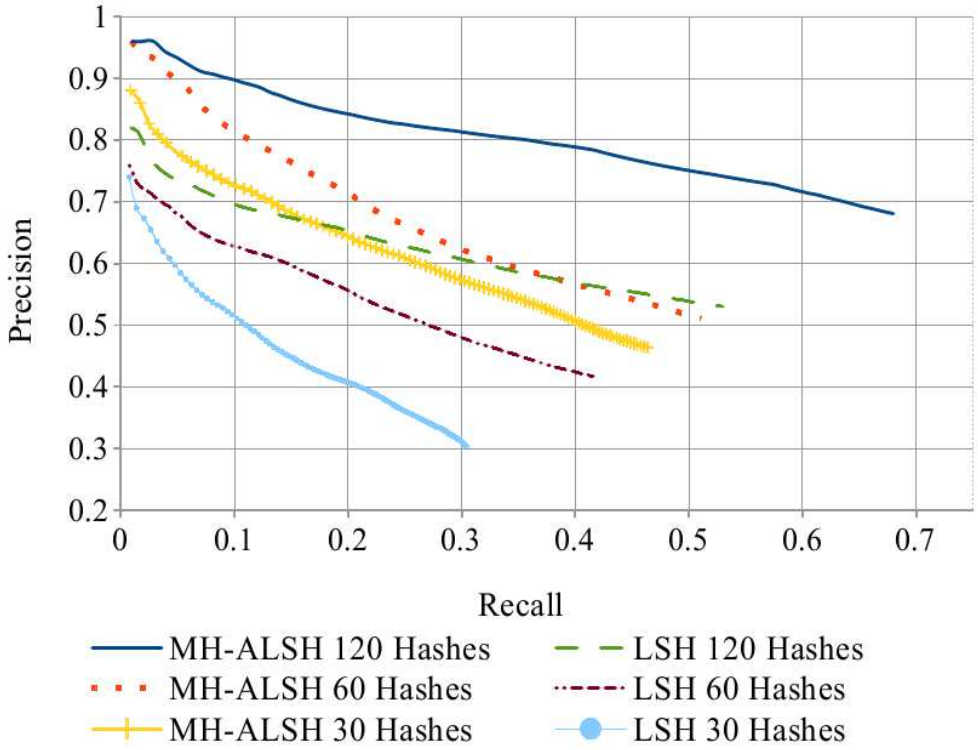

Figure 6.5: Top-100 retrieving precision and recall for MH-ALSH and LSH

of hashing functions from 30 to 120. In each setting, we adjust the parameters b and r to make sure the similarity threshold remains the same.

Figure 6.5 shows the performance (in terms of precision and recall) of MH-ALSH and LSH set expansion using Rank by Frequency Count (FC achieves the best accuracy performance in both cases). We can see that MH-ALSH is far more accurate than traditional LSH, which confirms the asymmetric modification for LSH does work better in set expansion contexts; and, with an increase in the number of hashing functions, the MH-ALSH performance can be improved further. This is because the longer the MinHash signature is, the more representative it stands for the original set. Of course, more hashing functions means the size of the indexes will increase, and the running time may also be affected.

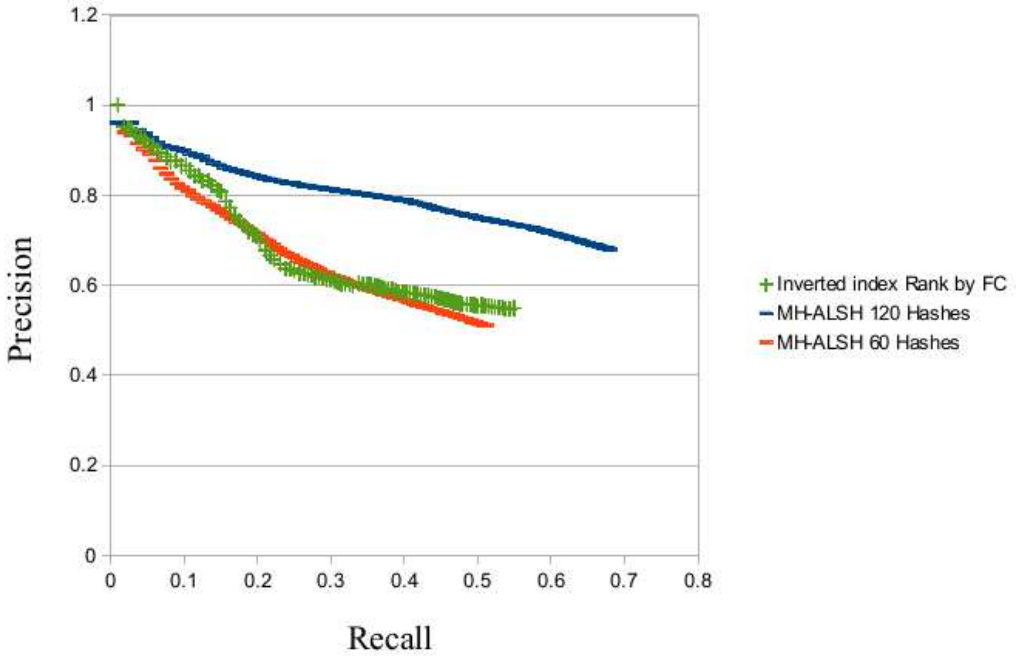

Figure 6.6: Top-100 retrieving precision and recall for MH-ALSH and Inverted index

6.3.5 Inverted index v.s. MH-ALSH

In Chapter 6.3.1, we have compared the timing performance of inverted index and MH-ALSH based set expansion systems. Now we want to show the accuracy performance comparison of two.

For inverted index, we choose the one with the best performance result which is achieved from Rank by Frequency Count. As for the MH-ALSH, we know that the more hashing functions, the better the performance will be. So we choose two typical parameter settings for comparison. One is MH-ALSH with 60 hashes, which consumes approximately the same memory space as the inverted index. Another one is MH-ALSH with 120 hashes, which takes twice as much memory as the inverted index. The result is shown in Figure 6.6.

From Figure 6.6, we can learn that the performance of MH-ALSH and inverted index are very close when they have the same amount of memory. However, when given more memory, MH-ALSH can have better accuracy per-

formance.

Chapter 7

Conclusions and Future Work

In this thesis, we have presented two efficient offline strategies for online set expansion, inverted index based and MH-ALSH based. Both of them have their own strengths and weaknesses. Inverted index based methods have a long latency when the post lists are very long, while MH-ALSH does not have this problem. On the other hand, inverted index is quite easy to setup for any datasets, while MH-ALSH needs carefully tuned parameters to achieve the best performance.

To our best knowledge, this is the first work on efficient set expansion for large data set collections. We evaluated our methods on real sets extracted from Wikipedia tables, which consists of near two million sets. Our proposed methods can also be extended to even larger scale datasets by incorporating more data sources, which could produce a set expansion system with huge potential applications.

7.1 Future work

After we use inverted index or MH-ALSH to retrieve relevant sets, there are still many potential ways to merge the sets and rank all the elements based on some scoring metrics. We have tried frequency count as our main merge and rank function. Integrating other rank and aggregation functions such as random walk is a possible future direction.

To further speedup the query response time, using some parallel processing techniques is also a promising direction to explore.

7.2 Future applications

Set expansion systems are of practical importance and can be used in various applications. For instance, web search engines may use the set expansion tools to create a comprehensive entity repository (for, say, brand names of each product category), in order to deliver better results to entity-oriented queries [He and Xin, 2011].

Another application for set expansion is recommendation systems. If we treat the few preferred products of a user as seed set, then set expansion may be used to find other potential products that may also meet the user's preference. In this context, the preferred products of every user makes a set which can be used in a set expansion.

Bibliography

Andrei Z Broder. On the resemblance and containment of documents. In *Proc. SEQUENCES'97*, pages 21–29. IEEE, 1997.

Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM'03*, pages 426–434. ACM, 2003.

David Carmel and Einat Amitay. Juru at trec 2006: Taat versus daat in the terabyte track. In *TREC'06*. Citeseer, 2006.

Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. STOC'14*, pages 380–388. ACM, 2002.

Surajit Chaudhuri, Kenneth Church, Arnd Christian König, and Liying Sui. Heavy-tailed distributions and multi-keyword queries. In *Proc. SIGIR'07*, pages 663–670. ACM, 2007.

J Shane Culpepper, Matthias Petri, and Falk Scholer. Efficient in-memory top-k document retrieval. In *Proc. SIGIR'12*, pages 225–234. ACM, 2012.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. SOCG'04*, pages 253–262. ACM, 2004.

Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proc. SIGIR'11*, pages 993–1002. ACM, 2011.

Marcus Fontoura, Vanja Josifovski, Jinhui Liu, Srihari Venkatesan, Xiangfei Zhu, and Jason Zien. Evaluation strategies for top-k queries over memory-resident inverted indexes. *VLDB Endowment*, 4(12):1213–1224, 2011.

Zoubin Ghahramani and Katherine A Heller. Bayesian sets. In *Proc. NIPS'05*, volume 2, pages 22–23, 2005.

Aristides Gionis, Dimitrios Gunopulos, and Nick Koudas. Efficient and tumble similar set retrieval. In *Proc. SIGMOD'01*, pages 247–258. ACM, 2001.

Yeye He and Dong Xin. Seisa: set expansion by iterative similarity aggregation. In *Proc. WWW'11*, pages 427–436. ACM, 2011.

Ihab F Ilyas, George Beskales, and Mohamed A Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

Simon Jonassen and Svein Erik Bratsberg. Efficient compressed inverted index skipping for disjunctive text-queries. In *Advances in Information Retrieval*, pages 530–542. Springer, 2011.

D Knuth. The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching, 1968.

Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

Nikos Mamoulis, David W Cheung, and Wang Lian. Similarity search in sets and categorical data using the signature tree. In *Proc. ICDE'03*, pages 75–86. IEEE, 2003.

Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. Web-scale distributional similarity and entity set expansion. In *Proc. EMNLP'09*, pages 938–947. ACL, 2009.

Manish Patil, Sharma V Thankachan, Rahul Shah, Wing-Kai Hon, Jeffrey Scott Vitter, and Sabrina Chandrasekaran. Inverted indexes for phrases and strings. In *Proc. of SIGIR'11*, pages 555–564. ACM, 2011.

Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*, volume 77. Cambridge University Press Cambridge, 2012.

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

Luis Sarmento, Valentin Jijkuon, Maarten de Rijke, and Eugenio Oliveira. More like these: growing entity classes from seeds. In *Proc. CIKM'07*, pages 959–962. ACM, 2007.

Anshumali Shrivastava and Ping Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Proc. NIPS'14*, pages 2321–2329, 2014.

Anshumali Shrivastava and Ping Li. Asymmetric minwise hashing for indexing binary inner products and set containment. In *Proc. WWW'15*. ACM, 2015.

Malcolm Slaney and Michael Casey. Locality-sensitive hashing for finding nearest neighbors [lecture notes]. *Signal Processing Magazine, IEEE*, 25(2): 128–131, 2008.

Richard C Wang and William W Cohen. Language-independent set expansion of named entities using the web. In *ICDM'07. Eighth IEEE International Conference on*, pages 342–350. IEEE, 2007.

Richard C Wang and William W Cohen. Iterative set expansion of named entities using the web. In *ICDM'08. Eighth IEEE International Conference on*, pages 1091–1096. IEEE, 2008.

Richard C Wang and William W Cohen. Character-level analysis of semi-structured documents for set expansion. In *Proc. EMNLP'09*, pages 1503–1512. ACL, 2009.

Ho Chung Wu, Robert Wing Pong Luk, Kam Fai Wong, and Kui Lam Kwok. Interpreting tf-idf term weights as making relevance decisions. *ACM Transactions on Information Systems*, 26(3):13, 2008.

Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW'09*, pages 401–410. ACM, 2009.

Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006.