AN ABSTRACT OF THE TECHNICAL REPORT OF

Abtin Khodadadi for the degree of Master of Science in Computer Science presented on September 3, 2019.

Title: Improving the Effectiveness of Answering Keyword Queries Using Effective Subsets

Abstract approved: _____

Arash Termehchy

Most database users do not know formal query languages, such as SQL, and prefer to express their information needs using usable query languages, such as keyword queries. Keyword queries, however, are inherently ambiguous and challenging for the database systems to understand and answer effectively. We propose a novel approach to improving the effectiveness of answering keyword queries by processing them over subsets of the database, called effective subsets. Effective subsets contain only the most promising candidate answers for these queries. We have shown that our method significantly improves the effectiveness of answering keyword queries using extensive empirical study.

Improving the Effectiveness of Answering Keyword Queries Using Effective Subsets

by

Abtin Khodadadi

A TECHNICAL REPORT

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 3, 2019
Commencement June 2020

Master of Science technical report of Abtin Khodadadi presented on September 3, 2019.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my technical report will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my technical report to any reader upon request.

Abtin Khodadadi, Author

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Dr. Arash Termehchy, for his support and guidance. Taking part in his research group, was a great chance for me to advance my knowledge and skills.

I acknowledge the immense contribution of Vahid Ghadakchi. This work is in good part in continuation to his research.

And as always, my special thanks goes to my family and friends. They are the sun that brightens my day – even in cloudy days of Corvallis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

Many users, such as scientists, are *not* familiar with (formal) query languages and concepts like schema [22]. Also, they often do *not* exactly know the schema and content of their databases. Thus, it is challenging for them to formulate their information needs over semi-structured and structured data-sets. To address this problem, researches have proposed keyword query interfaces (KQIs) over which a user can express a query simply as a set of keywords without any need to know any formal query languages and/or the schema of their databases [20, 9, 12]. As an example, consider the DBLP (*dblp.uni-trier.de*) database which contains information on computer science publications whose fragments are shown in Figure 1.1. Suppose that a user wants to find the papers on cluster data processing by *Sanjay Ghemawat*. These are the papers with IDs 01 and 03 in Figure 1.1. To retrieve these answers, the user may submit the keyword query q_1 : "cluster data processing sanjay" to retrieve these papers.

Since keyword queries do *not* generally express users' exact information needs,

ID	**Title**	**Author**	**Year**
01	MapReduce: simplified data processing on large clusters	Jeff Dean, Sanjay Ghemawat	2008
02	Enabling cross-platform data processing	D. Agrawal, Sanjay Chawla	2011
03	MapReduce: a flexible data processing tool	Jeff Dean, Sanjay Ghemawat	2010
04	Graph data processing on clusters	Sanjay Rakesh	2014
05	Secure data processing in clusters	Sanjay Balraj	2015
⋮	⋮	⋮	⋮

Figure 1.1: A Fragment of the DBLP Database

it is challenging for a KQI to satisfy the true information needs behind these queries [30, 12]. Generally speaking, the KQI finds the tuples in the database that contain the input keywords, ranks them according to some ranking function that measure how well each tuple matches the keywords in the query, and returns the ranked list to the user. For instance, in our example, as an answer to q_1 over the database in figure 1.1, the user may get a ranked list of papers with IDs 04, 05, 01 and 03, as all these records contain the keywords in q_1. Although all of the returned tuples contain the keywords in the query, only the last two, i.e., papers with IDs 01 and 03, are relevant to the input query.

Current KQIs often return too many non-relevant answers and suffer from low ranking quality over large databases [2, 7, 8, 14, 30]. Therefore, users often *cannot* find their desired information using these queries. Empirical evaluations of keyword query answering systems over semi-structured data indicate that most returned answers including the top-ranked ones are *not* relevant to the input query [2, 7, 8]. Similar results have been reported in the empirical evaluation of the KQIs over relational databases [14]. For example, in many cases, only 10%-20% of the returned answers are relevant to the input query [2, 7, 14].

Moreover, as KQIs have to examine a large number of possible matches and answers to the input keyword query, it takes a long time for them to answer users' queries [14, 6]. The query processing time is particularly time-consuming over relational databases [6]. For queries over relational databases, a KQI has to first find tuples in the base relations. Since none of the tuples in the base tables may be sufficiently relevant to the query and have a relatively low score, the

KQI has to compute all possible joins of these tuples across various base relations. Empirical studies show that it may take up to 200-400 seconds to process a keyword query over relational databases [6]. Since keyword queries may often be used in an interactive fashion to explore the database, users need a significantly shorter response time [12, 1].

It has been long established that in most information systems, query frequencies and their relevant answers follow a power law distribution [32, 34]. This assumption is the basis of our key intuition that there is a small subset of tuples in the database that contains many relevant answers to most queries. Because this subset has far fewer tuples than the entire database, the chance of making a mistake by KQI over this subset, i.e., returning a non-relevant answer, is less than doing so over the entire database [30]. Thus, on average, the KQI may return fewer non-relevant answers to queries than when it processes the queries over the entire database. Furthermore, since this subset is much smaller than the database, answering queries over the subset will be potentially much faster.

For example, assume that papers with IDs 01, 03, and 05 are more popular among users, i.e., they are relevant answers to more queries than the papers with IDs 02 and 04 in the database shown in Figure 1.1. One may run q_1 : "cluster data processing sanjay" over only these records and get a ranked list of papers with IDs 05, 01, and 03, which contains more relevant answer than the returned list of tuples over the entire database illustrated in Figure 1.1. As a matter of fact, our empirical results over several real-world query workloads confirm our key intuition.

The first challenge in enhancing the mentioned idea is to find such an *effective*

subset. If the subset contains too few tuples, it will not contain the relevant answers of the majority of the queries or it may contain only a small fraction of the relevant answers of most queries (small recall). On the other hand, if the subset contains too many tuples, then it will suffer from the same problems as running queries over the entire database. Thus, we should address how to pick an effective subset that contains many relevant answers to most queries.

Although an effective subset contains relevant answers of many queries, it will not contain any relevant answers to a small fraction of queries. Thus, the database system should identify these queries and use the full database to answer these queries.

In this report, we open the debate on using an effective subset of a large database to answer keyword queries over the database to increase their effectiveness. To the best of our knowledge, this approach has *not* been examined to improve the effectiveness of answering keyword queries over datasets. We show that using an effective subset, the KQI can significantly reduce the number of non-relevant answers in its results and reduce the query response time. Moreover, we show that by carefully selecting the tuples in the effective subset, one can also improve the recall of answering queries on average. The improvement of the recall is, in fact, an interesting result as one may expect otherwise. To further improve the effectiveness of answering queries, we propose a method that predicts whether a query can be answered more effectively on the subset or the entire database and forwards the query accordingly. One may increase the effectiveness of the keyword search by designing new search and ranking algorithms. Our proposed

approach is orthogonal to such methods and can be used with any of the keyword search algorithms to increase its effectiveness. To this end, we make the following contributions.

- We analyze the impact of using a subset of the entire database to answer keyword queries. Our results indicate that there are effective subsets such that, using only those subsets to answer queries, a KQI is able to improve the average ranking quality, average recall, or both for submitted queries (Chapter 3).

- As we discussed, the effective subset may not have all or some of the relevant answers to many queries. We propose a novel method to predict whether a query can be answered more effectively over the effective subset or the entire database. A KQI uses the result of this method to forward each input query to the effective subset or the entire dataset (Chapter 4).

- We provide a comprehensive empirical study of our method over multiple real-world large databases and query logs. Our results indicate that our approach substantially improves both precision and recall of answering keyword queries over large databases. They also show that our method to find the right subset of the dataset to answer the query significantly increases ranking quality and recall of answering queries (Chapter 5).

Chapter 2: Related Works

Existing approaches to keyword search over relational data-bases fall into two categories: graph-based systems and schema-based systems. Graph based methods convert the database into a data graph and perform the search on it [9, 23, 16, 19]. Schema based approaches consider the schema as a graph and directly search the relational database by generating and executing SQL queries [20, 21, 26, 29]. We refer the reader to [12] for a survey of keyword search approaches. Although the mentioned methods have high effectiveness and efficiency on small and medium size databases, most of them do not scale well to larger databases [13, 14]. Our proposed approach can be coupled with these search methods to increase the effectiveness of search over large databases.

In [6], the authors propose a keyword search method where the system quickly returns some answers to the user by scanning a part of the database, and generates forms to allow the user to explore the rest. Our approach is different because we aim to answer the queries in one shot without the need for further interactions.

Hawkin et al. [18] have studied the impact of collection size on information retrieval effectiveness. Their hypothesis states that precision@20 on a sample of a collection is less than precision@20 on the whole collection. This is because, in their experiment, the number of relevant answers over the sampled collection is less than the original collection. They provide a theoretical framework as well as

experimental results to justify this hypothesis and examine the causes of the drop in the search effectiveness. Furthermore, they state Document Frequency feature used in most retrieval methods varies over sample and original collection. In their experiments, they pick the subsets randomly, however, we pick the subsets based on user interaction history.

Search engines store large inverted indexes to answer users' queries. To reduce the inverted index size and query time, search engines prune their inverted index. The main objective of pruning is to reduce the size of the index as much as possible without changing the top ranked query results. Pruning techniques fall into two classes: keyword pruning and document pruning. In the first method, each term in the inverted index is assigned a score. The score can be computed based on IR scoring functions, access counts and information in the query log. Then, the keywords with low scores and their relevant postings are removed from the index. In the second approach, documents of each keyword are assigned a score and for each keyword, the documents with low scores are pruned [31]. Our approach is different than pruning in that its objective is to increases the search effectiveness whereas the pruning methods only focus on improving search efficiency while maintaining the search effectiveness. In fact, most of the pruning techniques sacrifice search effectiveness for its efficiency [4]. Furthermore, some IR systems use a two-tier index in which the first tier consists of a pruned index and the second tier is the original index. When a query is submitted to the system, the first batch of answers is computed based on the first tier of the index and the rest is computed based on the second tier. While this approach increases the efficiency of the search, it leads

to a degradation of the effectiveness [31]. In contrast, our system only uses one source and it does not combine the results of queries from different tiers/sources.

Caching techniques have been used in search engines [5, 10], database management systems and multi-tier client-server web-based applications [15, 28, 3, 25]. Our proposed framework has three major differences with a cache: 1) The goal of caching is solely to improve the efficiency of the search but the main objective of our framework is to increase the search effectiveness. 2) Size of a traditional cache is fixed and determined based on the available resources however the size of the effective subset does not depend on the available resources. In fact, finding the right size for the effective subset is one of the main challenges of using such systems. 3) A larger cache has a better overall performance but a larger subset does not always perform better than a smaller one.

Volume and velocity of big data makes its handling and analytical processing a costly process. To cope with these problems, a radical approach is to let the database semi-autonomously remove some of its data. Kersten et al. [24] have proposed a database with amnesia where tuples get forgotten based on different strategies. Their goal is to fix an upper bound for the database and yet be able to answer the submitted aggregate queries. Their work is different than ours as they are focused on numerical data and they do not intend to increase the accuracy of answering the queries.

Machine learning based ranking methods (a.k.a learn to rank methods) use prior probabilities as a feature to train their ranking models [27]. These prior probabilities are independent of any specific query and may be computed based

on the previous interactions with users or side information, e.g., PageRank scores. Our approach is different as we ignore the items with lower prior access count when searching for relevant answers of popular queries instead of using the access counts for ranking candidate answers.

Dong et al. [17] have studied the problem of picking a subset of data sources to optimize data fusion accuracy. Their problem is similar to ours as both of them are trying to discard a part of the data to achieve higher effectiveness or accuracy but there are fundamental differences between the two. In their setting, adding data sources is costly and data sources may have common information. But in our setting, adding data does not have a cost and the added data does not have any tuples in common with the existing data.

Chapter 3: Impact of Database Size on The Search Effectiveness

In this chapter, we analyze the impact of database size on search effectiveness. We focus on databases with a single relation. This can be a relational database with one table or a collection of semi-structured documents such as XML or JSON documents.

3.1 Empirical Study

There is an upper bound for the search effectiveness based on the database size. However, it remains an open question whether the provided bounds are tight enough to be used in practice. In this section, we answer this question by conducting extensive experiments on real-world datasets and query logs.

3.1.1 Datasets and Query Workloads

We conduct the empirical study using three datasets from Wikipedia, StackOverflow and MSLR-WEB30K. The Wikipedia dataset contains the information on 11.2 million Wikipedia articles[1]. Each article has a title and a body field. This dataset also contains users' access count for each article that is collected over a period of 3

[1]Available at: http://inex.mmci.uni-saarland.de/tracks/lod/2013/index.html

months[2] and we use them to compute data item popularities. For this dataset, we carry out the experiments on two query workloads with different characteristics. The first query workload is obtained from INEX Adhoc Track [8]. It is formed of 150 keyword queries and their relevant answers over Wikipedia. For each query, the number of relevant answers varies between 1 and 134. The second query workload is a sample of queries submitted to the Bing search engine. It contains more than 6000 keyword queries, most of which have a single relevant answer in Wikipedia. Note that these two query workloads and the access count of Wikipedia articles are collected independently. This is important because otherwise the data items that are relevant to a query in our query log will have a high popularity which will introduce a bias into the final results.

The StackOverflow dataset contains the information of StackOverflow questions and answers[3]. Each post in the StackOverflow website has a question and may have zero or one accepted answer. Using the questions and their accepted answer, we build a query workload for StackOverflow dataset. We pick the questions that have accepted answers in the dataset and use the title of the question as a keyword query. The final query workload contains 1 million queries and 1 million relevant answers. Furthermore, each post in StackOverflow has a view count that is the number of times a post has been viewed. We use this number to compute data item popularities and query frequencies. More precisely, if a question (or an accepted answer) has been visited a certain amount

[2]Available at http://dumps.wikimedia.org/other/analytics
[3]Available at: https://archive.org/download/stackexchange

of time, we set the frequency of the query (or the popularity of the accepted answer) to this number. We divide the view counts into two independent sets, one for queries and the other for the answers.

The MSLR-WEB30K dataset contains 30,000 queries sampled from Bing search engine and 3.7 million distinct URLs. Rows of this dataset are query-URL pairs. Each pair consists of query ID, URL ID and a 136 dimensional feature vector including query-URL click count. We use URL click counts to compute access counts of each URL. Furthermore, for each query, we use the maximum query-URL click count as the frequency of that query. More details on this dataset can be found in [33].

3.1.2 Implementation

We have implemented the experiments using Apache Lucene 6.5[4] with BM25 scoring method [30]. For the Wikipedia dataset where each article has a title and a body, we compute the relevance score of the document as a weighted sum of scores of its attributes. We find the optimal values of the weights using grid search. For each query, we retrieve the top k relevant tuples. We set the k = 20 for p@20 and MRR and k = 100 for recall. Some search engines use the access count of a web page as a feature in their scoring function to increase the effectiveness of the retrieval. This approach is called score boosting. We have tried boosting the retrieval system in our experiments and it did not have a significant improvement.

[4]https://lucene.apache.org/

Thus, we report the results of retrieval without any boosting techniques[30].

3.1.3 Experimental Environment

We run the experiments on a Linux server with 30 Intel(R) Xeon(R) 2.30GHz cores, 500GB of memory, 100 TB of disk space and CentOS 7 operating system. We have implemented the experiments using Java 1.8 and Python 3.6.4.

3.1.4 Building The Subset of The Database

We evaluate the effectiveness of query answering over subsets with different sizes. We build subsets of different sizes and compute the effectiveness using each subset. Given database I, let I_k be the subset of I that contains the top $k\%$ of the most popular tuples in the database. We build a sequence of subsets of I as $\{I_1 \dots I_{100}\}$. Given tuple $t \in I$, we denote the popularity of t as $w(t)$. The sequence of the subsets has the following characteristics:

1. $I_i \subset I_{i+1}$
2. $\forall t \in I_i, \forall t' \in I_{i+1} : w(t) \geq w(t')$

We submit queries of the different query workloads to each subset and report the results of each dataset.

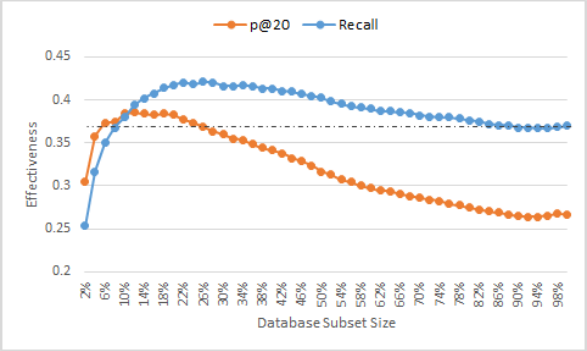

Figure 3.1: Effectiveness of answering INEX queries

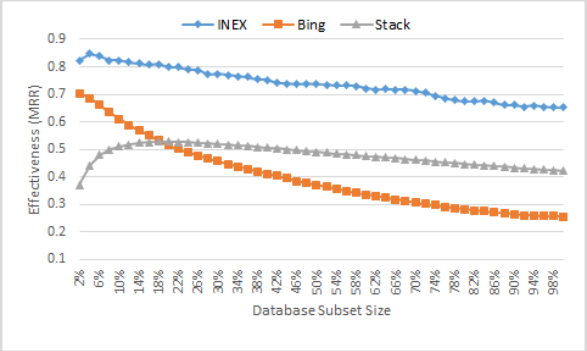

Figure 3.2: MRR of answering INEX, Bing and StackOverflow

3.1.5 Results of The Wikipedia Experiment

Figure 3.1 shows the effectiveness of answering INEX queries over subsets $I_1 \dots I_{100}$ of Wiki-pedia. The x axis shows the size of the subset as a fraction of the whole database and the y axis shows the average p@20 and *recall* of the queries. For very small subsets, the system has a low p@20 because these subsets do not contain enough relevant answers. As the size of the subset gets larger, p@20 increases until a certain point. After this point, even though increasing the size, adds more relevant answers to the subset, it increases the chance of making mistakes by the database and we see a decrease in the p@20. The same analysis holds for recall.

Figure 3.2 shows a similar experiment on Wikipedia using Bing queries. Most of these queries have a single relevant answer. Thus, we use the mean reciprocal rank of the results to measure the effectiveness of the search. For this query workload, I_2 has the highest MRR and for subsets larger than I_2, MRR has a decreasing trend.

3.1.6 Results of The StackOverflow Experiment

Figure 3.2 shows the effectiveness of query answering over different subsets of the StackOverflow dataset. The subset with 18% of the data has the highest effectiveness. For larger subsets, the effectiveness gradually decreases. In this experiment, there is a one-to-one mapping between the queries and their answers. Thus, excluding one answer from a subset will result in zero relevant answers for its corresponding query. More precisely, the effective subset with 18% of the data only

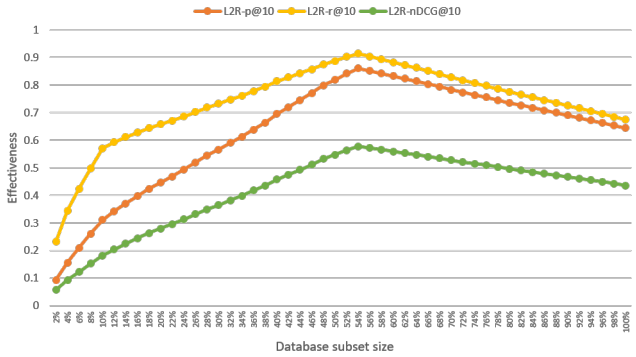

Figure 3.3: Effectiveness of MSLR queries

contains the relevant answers of 18% of the queries. However, these queries are submitted so frequently that on average, the subset achieves higher effectiveness than the full collection.

3.1.7 Results of the MSLR-WEB30K Experiment

Figure 3.3 shows P@10 and recall@10 and nDCG@10 of query answering over different subsets of MSLR dataset. The results show that the subset of URLs with 54% of the URLs achieves the highest search effectiveness. In this experiment, the effective subset is larger than the previous cases. Also, the decrease in the effectiveness is not as intense. One reason for this is that the URLs of MSLR dataset are sampled from results of Bing and for each query more than 54% of the URLs are relevant answers. However, in a regular collection, number of relevant answers for each query is a very small percentage of the whole collection.

These experiments show that, given a database, if the size of the database

grows larger than a threshold, the effectiveness of query answering will drop. As the database gets larger, the decrease in effectiveness becomes more significant. In the next section, we use these results to build a subset of the database that delivers significantly higher effectiveness in answering queries.

Chapter 4: Improving The Effectiveness of Answering Infrequent Queries

In this chapter, we present an approaches to improve the search effectiveness of the infrequent queries. We develop a methods that, given the subset and full database, predict which one of these data sources deliver a higher search effectiveness. If the models predict that the full database has a higher search effectiveness, then the query is classified/labeled as infrequent. The queries that are labeled as infrequent, are submitted to the full database rather than the subset.

4.1 Detecting Infrequent Queries using Machine Learning

In this section, we present a method to train a logistic regression classifier that predicts if a query is infrequent or not. Each query is represented by a feature vector. We extract the features over the subset and the rest of the database i.e. database excluding the subset. We present three sets of features that are used in our system and explain why each group is useful for building the classifier.

4.1.1 Content-Based Features

Content-based features are based on the probability distribution of words in the given database. Query likelihood score explained in the previous section is one of the content-based features. Some other examples of these features are as follows:

Covered term ratio: is the fraction of the terms in the query that appear in a data source. If a query has a higher covered term ratio over the subset compared to the rest of the database, answering this query over the subset will return relevant results with a higher likelihood. For example, consider a user that is looking for Michael Stonebraker's paper on VoltDB and submits query `stonebraker voltDB`. If the subset contains the VoltDB paper, the subset has covered term ratio = 1. Now, if the rest of the database contains other papers of Stonebraker which are not about VoltDB, the covered term ratio of the rest of the database for the given query will be $\frac{1}{2}$. In this case, the subset has a better coverage than the rest of the database which means the query is not likely to be infrequent. However, if the VoltDB paper is included in the rest of the database, the feature will have a higher value over the rest of the database compared to the subset and with a higher chance, the query is infrequent.

Tuple Frequency: is the number of the tuples that a term appears in. Assume a user who is looking for papers of Stonebraker and submits the query `Stonebraker`. Let's assume the subset contains 50 papers by Stonebraker and the rest of the database contains 5. In this case, Tuple Frequency can be a good signal that the database should use the subset to answer the query. For queries with more than

one term, the aggregate tuple frequency of the terms is used as the final value of the feature. We use different aggregate functions such as average tuple frequency of terms of the query.

Most of the content-based features are defined based on the terms of the query. We extract the same features for bi-words of the query as well. For example, given query `data processing` and feature Tuple Frequency, we extract the tuple frequency of the term `data`, processing and also the tuple frequency of the bi-word `data processing`.

4.1.2 Popularity-Based Features

One of the major distinguishing factors of the subset from the rest of the database is the popularity of the tuples in them. More precisely, any tuple that has a higher popularity than a certain threshold is included in the subset. We use this characteristic of the subset to design a second set of features which reflects the popularity of the relevant answers of a query. Inspired by the language model approach, we design a popularity model which is a statistical model of the popularity of the terms in a database. For each term in the database, we compute two popularity statistics: 1) The average popularity of the tuples containing that term. 2) The minimum popularity of the tuples containing that term. We use these two statistics to estimate the popularity of terms of a query. Then we aggregate the popularities of all query terms into a single value that estimates the popularity of the relevant answers of that query. For aggregation, we use minimum and average functions.

Consider a user that is looking for papers on data processing using MapReduce and submits `map-reduce framework`. The term `framework` can happen in tuples with different popularities thus its popularity is 0.45 whereas the term `MapReduce` happens in the tuples with high popularity and it's popularity is 0.85. The average popularity of these two terms is 0.65 which is an indicator that most of the relevant answers of this query can be popular, thus query is not likely to be infrequent. Similar to content-based features, we extract popularity features for terms as well as bi-words of the query.

4.1.3 Query Difficulty Based Features

IR researchers have developed query difficulty metrics to predict the quality of the search results of a query [11]. Given a query and a data source, these methods compute a number that indicates the hardness of a query. These metrics can be applied to our problem to extract further features. Let us say the user submits query q where its difficulty metric over the full database is a value close to zero. This is an indicator that answering this query over the full database is *easy* and will result in high search effectiveness. In this case, it is reasonable to use the full database rather than the subset. However, if the estimated query difficulty is high over the full database, it means the quality of the search over the full database is likely to be low and one may consider submitting it to the subset. We use different difficulty metrics such as Clarity Score, Collection Query Similarity, etc [11]. We only include the difficulty metrics that can be computed for a query

without actually conducting the search. There are other difficulty metrics that are computed based on the search results, however, using those metrics in our system would be inefficient as it doubles the search time. More precisely, to use those features, one should conduct the search twice, once to compute the metric and classify the query and second time to conduct the search on the subset or full database based on the results of the classifier.

4.1.4 Training The Infrequent Query Classifier

We use the logistic regression method to train our classifier. Logistic regression is a good fit for this problem because of the following reasons. First, it has higher interpretability and it is easier to see which features have a higher impact on the classification decision. Second, when the signal-to-noise ratio is low, logistic regression usually outperforms other methods. To train the classifier, we use a sample of the query workload. To build the training data, we submit each query in the sample once to the subset and once to the full database. If the search effectiveness over the full database is larger than the subset, we label the query as infrequent. Otherwise, it is labeled as a popular query. We extract 36 features per each field of the database. Most of the features mentioned above are extracted once over the subset as f_s and once over the rest of the database as f_r. A comparison of these two features can be an indicator of the class of the query. Since logistic regression is a linear model, it does not consider the non-linear comparison of these features. To include non-linear

comparison of these features, we add division of them defined as $\frac{f_s}{f_r}$. These extra features represent the multitude of the difference between features.

The final classifier is trained using the extracted features and their non-linear combinations. Using this classifier, we are able to predict the type of query prior to the search and submit the infrequent queries to the full database. We evaluate the effectiveness of this system in Chapter 5. Furthermore, we show the overhead of using a classifier prior to search is negligible compared to the search time. This is because the features are extracted using the pre-built indexes on the database. Also applying logistic regression classifier to a feature vector is very fast. The detailed performance evaluation of this system is presented in Chapter 5.

Chapter 5: Experiments

In this section, first, we evaluate the effectiveness of the subsets we have built based on exhaustive search. Then, we evaluate the effectiveness of query answering using our system. Furthermore, we evaluate the accuracy of the infrequent query detection method presented in Chapter 4.

5.1 Experiment Setting

We use the normalized forms of Wikipedia and StackOverflow databases introduced in Section 3.1. The details of these datasets are shown in Table 5.1. The Wikipedia database contains 5 tables: article, article-link, link, article-image and image stored in a MySQL database. The indexed text attributes used for search are *article.body*, *image.caption* and *link.url*. This dataset contains access counts for articles, images, and links. The StackOverflow dataset contains the information of StackOverflow posts with the following tables: *posts, post-comment, comments, post-tag, tags* and their access counts. The attributes used for search are *posts.text*, *tags.tag_names* and *comments.body*. We store these databases in a MySQL 5.1 engine. The query workloads used in this section are the same as Section 3.1.

To create the tuple sets with relevance score we use Apache Lucene and

Table 5.1: Dataset Information

Dataset	#Tuples	#Relations	Size (GB)
Wikipedia	130M	5	35
StackOverflow	304M	5	2.3

Table 5.2: Evaluating the built subset against full database

Experiment	Effectiveness	
	Subset	DB
INEX-p@20	0.33	0.22
INEX-*rec*	0.29	0.22
Bing	0.51	0.08
StackOverflow	0.51	0.38

BM25 scoring technique [30]. We limit the size of the generated tuple sets based on a fraction of their max score. For example, if the highest score in a tuple set is s, we remove all the tuples with a score less than $\frac{s}{2}$ from the tuple set. This helps the IRStyle method to process the queries a reasonable time. For the experiment on p@20 and MRR, we retrieve the top 20 tuples and for the recall we retrieve the top 100 tuples. The experiment environment is similar to Section 3.1.

5.2 Evaluation of The Effective Subset

In this section, we evaluate the effectiveness of our subset estimator method. Given a database and a query workload, we randomly select 20% of the queries as training queries and keep the rest for testing. For INEX queries, we run the experiment once to maximize the p@20 and once to maximize the recall. For Bing and StackOverflow we run the algorithm with MRR as the effectiveness function. We execute

the test queries using IRStyle search method explained above once over the full database as the baseline and once over the effective subsets. For INEX experiment we report precision-at-20 (p@20) and recall as the effectiveness metrics and for Wikipedia-Bing and StackOverflow, we report MRR (as the queries of these experiments have one relevant answer).

The results of this experiment are shown in table 5.2. The rows are associated with experiments and the columns are the results of that experiment. As shown in the table, the subset delivers higher effectiveness than the baseline in all four experiments. The highest gain happens in the Bing experiment. This is because for the Bing experiment, the effective subset is much smaller (2%) and as discussed in Chapter 3, a smaller subset results in much fewer search mistakes by the database system. Furthermore, the effective subset for the recall has the largest size as explained in Chapter 3.

5.3 Evaluating The Infrequent Query Detection

In this section, we evaluate the query type prediction method. The objective of query type prediction is to detect the infrequent queries and improve their results while maintaining high average effectiveness for all queries. We present the effectiveness of query answering using the two infrequent query detection methods and compare it with the cases that we do not use this approach. Following is a list of different settings used for evaluating the infrequent query detection method:

- Subset: Using the effective subset to answer all queries

Table 5.3: Results of answering Bing Queries

Experiment	MRR		
	Popular	Infrequent	All
Subset	0.53	0.03	0.51
Database	0.07	0.51	0.08
ML	0.48	0.28	0.50
Best	0.53	0.51	0.53

- Database: Using the database to answer all queries
- ML: Using the logistic regression model to predict infrequent queries and reroute them to the database
- Best: Using an Oracle that knows the exact type of the query and routes the infrequent queries to the full database

To simulate the Oracle, we submit the query to both database and the subset and pick the results with higher effectiveness. The result of using the Oracle shows the best possible effectiveness that one can achieve. We carry out the evaluations on different datasets as before.

In the first experiment, the effective subset is built over Wikipedia using Bing train queries, and we train the logistic regression model as explained in Chapter 4. The accuracy of this model is 0.83. Then we use the test queries to evaluate the machine learning based infrequent query detection method. The result of this experiment is shown in Table 5.3. The columns of the table show the search effectiveness (MRR) of popular queries, infrequent queries, and all queries as well as the average running time of all queries in seconds. The rows indicate different

Table 5.4: Results of answering StackOverflow queries

Experiment	MRR		
	Popular	Infrequent	All
Subset	0.56	0.01	0.51
Database	0.36	0.50	0.38
ML	0.55	0.29	0.49
Best	0.56	0.50	0.55

settings related to each system. For all queries, the subset outperforms all other methods. However, it has a very low MRR of 0.03 for infrequent queries. The ML method has high effectiveness for all queries (0.50) and it increases the MRR of infrequent queries from 0.03 on subset to 0.28.

Next, we evaluate our system using the StackOverflow dataset using a similar approach as above. The results of this experiment are shown in Table 5.4. Similar to the previous experiment, the system that only uses the subset achieves the highest MRR for all queries. However, it suffers from low MRR on bad queries. The system that uses the full database has an opposite performance and finally the machine learning based infrequent query detection method is able to increase the effectiveness of infrequent queries from 0.01 to 0.29 while maintaining a high MRR for all queries.

Chapter 6: Conclusion

The objective of this report was to demonstrate the limitations of current keyword query systems over large databases and propose a method to improve these boundaries. Our main idea is to enhance user interaction information to identify a hot subset of the database, build a system based on this subset and use machine learning to utilize it in a keyword query system. Experimental results of evaluating this approach indicates that it is successful in increasing the effectiveness of the keyword search systems.

Bibliography

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 29–42, New York, NY, USA, 2013. ACM.

[2] James Allan, Donna Harman, Evangelos Kanoulas, Dan Li, Christophe Van Gysel, and Ellen Vorhees. Trec 2017 common core track overview. In *Proc. TREC*, 2017.

[3] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, Chandrasekaran Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 718–729. VLDB Endowment, 2003.

[4] Ismail S Altingovde, Rifat Ozcan, and Özgür Ulusoy. Static index pruning in web search engines: Combining term and document popularities with query views. *ACM Transactions on Information Systems (TOIS)*, 30(1):2, 2012.

[5] Ricardo Baeza-Yates, Aristides Gionis, Flavio Junqueira, Vanessa Murdock, Vassilis Plachouras, and Fabrizio Silvestri. The impact of caching on search engines. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 183–190. ACM, 2007.

[6] Akanksha Baid, Ian Rae, Jiexing Li, AnHai Doan, and Jeffrey Naughton. Toward scalable keyword search over relational data. *Proceedings of the VLDB Endowment*, 3(1-2):140–149, 2010.

[7] Patrice Bellot, Toine Bogers, Shlomo Geva, Mark Hall, Hugo Huurdeman, Jaap Kamps, Gabriella Kazai, Marijn Koolen, Véronique Moriceau, Josiane Mothe, Michael Preminger, Eric SanJuan, Ralf Schenkel, Mette Skov, Xavier Tannier, and David Walsh. Overview of inex 2014. In Evangelos Kanoulas,

Mihai Lupu, Paul Clough, Mark Sanderson, Mark Hall, Allan Hanbury, and Elaine Toms, editors, *Information Access Evaluation. Multilinguality, Multimodality, and Interaction*, pages 212–228, 2014.

[8] Patrice Bellot, Antoine Doucet, Shlomo Geva, Sairam Gurajada, Jaap Kamps, Gabriella Kazai, Marijn Koolen, Arunav Mishra, Véronique Moriceau, Josiane Mothe, et al. Overview of inex 2013. In *International Conference of the Cross-Language Evaluation Forum for European Languages*, pages 269–281. Springer, 2013.

[9] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using banks. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 431–440. IEEE, 2002.

[10] Berkant Barla Cambazoglu, Flavio P Junqueira, Vassilis Plachouras, Scott Banachowski, Baoqiu Cui, Swee Lim, and Bill Bridge. A refreshing perspective of search engine caching. In *Proceedings of the 19th international conference on World wide web*, pages 181–190. ACM, 2010.

[11] David Carmel and Elad Yom-Tov. Estimating the query difficulty for information retrieval. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, 2(1):1–89, 2010.

[12] Yi Chen, Wei Wang, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 1005–1010. ACM, 2009.

[13] Joel Coffman and Alfred C Weaver. A framework for evaluating database keyword search strategies. In *In CIKM*, pages 729–738. ACM, 2010.

[14] Joel Coffman and Alfred C Weaver. An empirical performance evaluation of relational keyword search techniques. *IEEE Transactions on Knowledge and Data Engineering*, 26(1):30–42, 2014.

[15] Shaul Dar, Michael J Franklin, Bjorn T Jonsson, Divesh Srivastava, Michael Tan, et al. Semantic data caching and replacement. In *VLDB*, volume 96, pages 330–341, 1996.

[16] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 836–845. IEEE, 2007.

[17] Xin Luna Dong, Barna Saha, and Divesh Srivastava. Less is more: Selecting sources wisely for integration. In *Proceedings of the VLDB Endowment*, volume 6, pages 37–48. VLDB Endowment, 2012.

[18] David Hawking and Stephen Robertson. On collection size and retrieval effectiveness. *Information retrieval*, 6(1):99–105, 2003.

[19] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 305–316. ACM, 2007.

[20] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 670–681. Elsevier, 2002.

[21] Vagelis Hristidis, Yannis Papakonstantinou, and Luis Gravano. -efficient ir-style keyword search over relational databases. In *PVLDB*, pages 850–861. Elsevier, 2003.

[22] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD*, 2007.

[23] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases*, pages 505–516. VLDB Endowment, 2005.

[24] Martin L. Kersten and Lefteris Sidirourgos. A database system with amnesia. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[25] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[26] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2006.

[27] Tie-Yan Liu. *Learning to rank for information retrieval*. Springer Science & Business Media, 2011.

[28] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611. ACM, 2002.

[29] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 115–126. ACM, 2007.

[30] Christopher Manning, Prabhakar Raghavan, and Hinrich Schutze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.

[31] Alexandros Ntoulas and Junghoo Cho. Pruning policies for two-tiered inverted index with correctness guarantee. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 191–198. ACM, 2007.

[32] Casper Petersen, Jakob Grue Simonsen, and Christina Lioma. Power law distributions in information retrieval. *ACM Transactions on Information Systems (TOIS)*, 34(2):8, 2016.

[33] Tao Qin and Tie-Yan Liu. Introducing LETOR 4.0 datasets. *CoRR*, abs/1306.2597, 2013.

[34] Ting Yao, Min Zhang, Yiqun Liu, Shaoping Ma, and Liyun Ru. Empirical study on rare query characteristics. In *IEEE/WIC/ACM International Conferences on Web Intelligence*, pages 7–14. IEEE Computer Society, 2011.