Less Data Delivers Higher Effectiveness for Keyword Queries

Vahid Ghadakchi
School of EECS
Oregon State University
ghadakcv@oregonstate.com

Abtin Khodadadi
School of EECS
Oregon State University
khodadaa@oregonstate.com

Arash Termehchy
School of EECS
Oregon State University
termehca@oregonstate.com

ABSTRACT

As many users, such as scientists, do *not* know the schema and/or content of their databases, they cannot precisely formulate their information needs using formal query languages, such as SQL. To help these users, researchers have proposed keyword query interfaces over which users can submit their information need using a set of keywords without the precise knowledge about the schema or content of the database. Despite their usability, keyword query interfaces suffer from low effectiveness in answering queries. Therefore, they may return many non-relevant answers or do *not* return many answers related to the input queries. It is well established that the effectiveness of answering queries decreases as the size of the dataset grows, given all other conditions are the same. In this paper, we propose an approach that uses only a relatively small subset of the database to answer most queries effectively. Since this subset may *not* contain the relevant answers to many queries, we also propose a method that predicts whether a query can be answered more effectively using this subset or the entire database. Our comprehensive empirical studies using multiple real-world databases and query workloads indicate that our approach significantly improves both the effectiveness and efficiency of answering queries.

CCS CONCEPTS

• **Information systems** → *Data management systems*; *Novelty in information retrieval.*

KEYWORDS

keyword query interfaces, effective keyword search

ACM Reference Format:
Vahid Ghadakchi, Abtin Khodadadi, and Arash Termehchy. 2019. Less Data Delivers Higher Effectiveness for Keyword Queries. In *31st International Conference on Scientific and Statistical Database Management (SSDBM '19), July 23–25, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3335783.3335794

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 [9, 12, 20]. 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. 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. 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, it is challenging for a KQI to satisfy the true information needs behind these queries [12, 31]. 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, 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, 31]. 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 [6, 14]. 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

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: A Fragment of the DBLP Database

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 [1, 12].

It has been long established that in most information systems, query frequencies and their relevant answers follow a power law distribution [33, 36]. 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 [31]. 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. 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. 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 paper, we open the debate on using an effective subset of a large database to answer keyword queries over the database to increase their effectiveness and efficiency. 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 and efficiency 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 and efficiency. To this end, we make the following contributions.

- We analyze the impact of using a subset of the entire database to answer keyword queries both theoretically and empirically. 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 (Section 2).
- We show how a KQI can utilize users' past interactions with the data to build the aforementioned effective subsets (Section 2.3).
- 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 (Section 3).
- We discuss and address the challenges of using our approach over relational data and address them (Section 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, recall, and efficiency 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 (Section 5).

2 IMPACT OF DATABASE SIZE ON THE SEARCH EFFECTIVENESS

In this section, 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. We extend the results of this section to databases with multiple relations in Section 4.2.

2.1 Theoretical Analysis

Consider a database instance I that contains information on n number of publications. Let Q be the set of all queries a user can submit to retrieve any paper. If the relevant answer of a query $q \in Q$ is in I, then I can potentially answer the query q, otherwise, it returns no relevant results. Let us build database J by adding more papers to I. Since J contains information on more papers, it can potentially answer more queries of Q. Hence, it is commonly believed that the average effectiveness of J on answering queries of Q is higher than I. While this belief is true for answering exact queries such as SQL queries, it does not hold for keyword queries. As shown in the example of Section 1, keyword queries are not the exact formulation of users' information needs, thus, databases may make mistakes in returning relevant answers of keyword queries. As the size of the database gets larger, the chance of making a mistake by database and returning a non-relevant answer increases [34].

On one side, adding more entities to the database increases the number of queries that can be answered by the database. On the other side, as the database gets larger, the chance of making a mistake by the database and returning a non-relevant answer to a query increases . Thus, it is not clear how does adding more entities to the database impacts the overall search effectiveness. To answer this question, we present a theoretical analysis of the problem. Later in this section, we verify the theoretical results by conducting empirical studies.

Consider database instance I and random query q over I. Let Q be the domain of q, $Pr(q)$ be its probability distribution, $q(I)$ be the returned results of q over I and $rel(q)$ be the set of its relevant answers. One of the metrics used to measure the search effectiveness of a top-k retrieval system is Precision-at-k ($p@k$). Precision-at-k of a random query q with distribution θ over I and its expected value is defined as:

$$p@k(q, I) = \frac{|q(I) \cap rel(q)|}{k},$$
$$\mathbb{E}[p@k(q, I)] = \sum_{q \in Q} Pr_\theta(q) p@k(q, I)$$

Access count of tuple t is the number of times that tuple has been accessed by users through queries or any other interactions. Given tuple $t \in I$, we define the popularity of t, denoted by $w(t)$, as the probability of t being a relevant answer to some query q. More precisely, given a random query q and the set of its relevant answers $rel(q)$, $w(t) = \sum_{q:t \in rel(q)} Pr(q)$. We estimate $w(t)$ as the normalized access count of t over sum of access counts of all tuples. In the example of Figure 1, each paper is a tuple and its popularity is computed as the number of times the paper has been accessed divided by the sum of all access counts. We define $I(m)$ as a subset of the database I that contains m most popular tuples of I.

In most database systems, access counts to tuples follow a power low distribution [36]. The following theorem states that, if $w(t)$ has a power law distribution then increasing the size of $I(m)$ beyond a certain point decreases the upper bound of search effectiveness over $I(m)$.

Theorem 2.1. *Consider database I such that for $t \in I$, $w(t)$ has a power law distribution. There is m_0 such that if $|I| > m_0$, for $m > m_0$, $\mathbb{E}[p@k(q, I(m))]$ is bounded above by a decreasing function of m.*

Proof. Consider random query q with distribution θ. Given tuple t, let q_t be a random query such that t is one of its relevant answers. The distribution of q_t is denoted by θ_t. Let $\mathbb{I}$ be the indicator function such that $\mathbb{I}(t \in q(I(m)))$ is one if $t \in q(I(m))$ and zero otherwise.

$$\begin{aligned}\mathbb{E}[p@k(q, I(m))] &= \sum_q Pr_\theta(q) \frac{|q(I(m)) \cap rel(q)|}{k} \\ &= \frac{1}{k} \sum_q Pr_\theta(q) \sum_{t:t \in rel(q)} \mathbb{I}(t \in q(I(m))) \\ &= \frac{1}{k} \sum_{t \in I(m)} \sum_{q_t} Pr_\theta(q) \mathbb{I}(t \in q(I(m)))\end{aligned}$$

Let $w(t) = \sum_{q_t} Pr_\theta(q_t)$, then $Pr_{\theta_t}(q_t) = Pr_\theta(q)/w(t)$ and we have:

$$\begin{aligned}\mathbb{E}[p@k(q, I(m))] &= \frac{1}{k} \sum_{t \in I(m)} w(t) \sum_{q_t} (Pr_\theta(q_t)/w(t)) \mathbb{I}(t \in q_t(I(m))) \\ &= \frac{1}{k} \sum_{t \in I(m)} w(t) \sum_{q_t} Pr_{\theta_t}(q_t) \mathbb{I}(t \in q_t(I(m))) \\ &= \frac{1}{k} \sum_{t \in I(m)} w(t) \mathbb{E}_{\theta_t}[\mathbb{I}(t \in q_t(I(m)))]\end{aligned}$$

In the above equation, $\mathbb{I}(t \in q(I))$ is a Bernoulli random variable and we can replace $\mathbb{E}[t \in q_t(I(m))]$ with $Pr(t \in q_t(I(m)))$. Let ϵ_t be the probability that t is ranked higher than another tuple in $q(I(m))$, then $Pr(t \in q(I(m)))$, the probability that t is in top-k retrieved tuples, is equal to ϵ_t^{m-k}.

$$\begin{aligned}\mathbb{E}[p@k(q, I(m))] &= \frac{1}{k} \sum_t w(t) Pr(t \in q(I)) \\ &= \frac{1}{k} \sum_t w(t) \epsilon_t^{m-k} \leq \max_t \{\epsilon_t^{(m-k)}\} \frac{1}{k} \sum_t w(t)\end{aligned}$$

Let r_t be the rank of tuple t based on its popularity $w(t)$. Given that the popularities follow a power law distribution we have $w(t) = \frac{1}{H_\alpha} \frac{1}{r_t^\alpha}$ where α is a real number greater than 1 and H_α is the m^{th} generalized harmonic number that is used for normalization of the probabilities. We can compute an upper bound for $\sum_t w(t)$ by integrating over values of r_t as follows:

$$\sum_{r_t=1}^{m} \frac{1}{r_t^\alpha} \leq 1 + \int_1^{m-1} \frac{1}{x^\alpha} dx = \frac{2(m-1)^{\alpha-1} - 1}{(m-1)^{\alpha-1}}$$

Using the above simplification we have:

$$\mathbb{E}[p@k(q, I(m))] \leq \max_t \{\epsilon_t^{(m-k)}\} \frac{1}{k} \frac{1}{H_\alpha} \frac{2(m-1)^{\alpha-1} - 1}{(m-1)^{\alpha-1}}$$

Let $\epsilon = \max_t \{\epsilon_t\}$. We compute the derivative of the above formula and factor out the constants:

$$\frac{\partial \mathbb{E}}{\partial m} = \frac{\epsilon^{m-k} \ln(\epsilon)\, m}{m+1} + \frac{\epsilon^{m-k}}{m+1} - \frac{\epsilon^{m-k} m}{(m+1)^2}$$

This derivative has a positive root at:

$$m_0 = \frac{\sqrt{\ln^2(\epsilon) - 4\ln(\epsilon)} - \ln(\epsilon)}{2\ln(\epsilon)}$$

For $m > m_0$, the derivative has a negative value which entails that for $m > m_0$ the function is strictly decreasing. Thus, if $|I| > m_0$, then for $m > m_0$, $\mathbb{E}[p@k(q, I(m))]$ is bounded by a decreasing function of m. □

This result shows that, if a database is sufficiently large, there is a subset of the database such that the highest achievable expected P@K over this subset is larger than the full database. This is because the mentioned subset is able to deliver higher effectiveness for tuples that are queried very often in the price of sacrificing the tuples that are not frequently queried.

Next, we investigate the impact of database size on recall. Recall of query q over database I, denoted by $rec(q, I)$, is the fraction of relevant answers returned by the database system:

$$rec(q, I) = \frac{|q(I) \cap rel(q)|}{|rel(q)|}$$
$$\mathbb{E}[rec(q, I)] = \sum_q Pr_\theta(q) rec(q, I)$$

Following theorem extends the results of Theorem 2.1 to the recall of answering queries over a database.

Theorem 2.2. *Consider database I such that for $t \in I$, $w(t)$ has a power law distribution. There is threshold m_1 such that if $|I| > m_1$, for $m > m_1$, $\mathbb{E}[rec(q, I(m))]$ is bounded above by a decreasing function of m.*

Proof. Similar to the previous proof:

$$\mathbb{E}[rec(q, I(m))] = \sum_q Pr_\theta(q) \frac{|q(I(m)) \cap rel(q)|}{|rel(q)|}$$
$$= \sum_q Pr_\theta(q) \frac{1}{|rel(q)|} \sum_{t \in rel(q)} \mathbb{I}(t \in q(I(m)))$$

Assuming that each tuple in the database gets at least one query and at most k' queries then $\frac{1}{k'} \leq \frac{1}{|rel(q)|} \leq 1$. Thus we have:

$$\mathbb{E}[rec(q(I))] \leq \sum_q Pr_\theta(q) \sum_{t \in rel(q)} \mathbb{I}(t \in q(I(m)))$$

The rest of the proof is similar to the proof of Theorem 2.1. □

This result shows that, if a database is sufficiently large, there is a subset of the database such that the highest achievable expected recall over this subset is larger than the full database. Note that the threshold m_1, which is used in this theorem, can be different than the threshold m_0 of Theorem 2.1. In fact, in most cases, m_1 is expected to have a larger value than m_0. We will discuss this in more detail in Section 2.2.

The last metric we examine is reciprocal-rank. Reciprocal rank (R-Rank) of query q over I is calculated as $\frac{1}{r}$ where r refers to the rank position of the first relevant answer in $q(I)$. Mean reciprocal rank (MRR) of queries Q over a database I is defined as the average of the reciprocal ranks of the queries in Q. Since the queries in our problem have different probabilities, we use the expected value of the R-Ranks of the queries to compute MRR:

$$MRR = \sum_q Pr(q) R\text{-}Rank(q, I)$$

This metric is useful when the queries have a single relevant answer. Using a similar approach to Theorem 2.1 and 2.2, it is easy to show similar results for MRR. The general idea here is to expand the R-Rank using the probabilistic approach presented in the proof of Theorem 2.1. For top-k results, the R-Rank can be expanded as:

$$R - Rank(q) = \sum_{i=1}^{k} \epsilon^{m-k+i} \frac{1}{i} \leq k\epsilon^{m-k}$$

Using this expansion, one can show that, if a database is sufficiently large, there is a subset of the database such that the highest achievable MRR over this subset is larger than the full database.

One of the factors that impacts the value of m_0 of Theorem 2.1 and m_1 of Theorem 2.2 is the similarity of the tuples in the database. If the tuples in a database are not similar, then the probability of making a mistake by retrieval system (i.e. returning a non-relevant tuple as an answer) decreases. In its extreme case, if the similarity between tuples is minimum, then the database system returns the correct answers with a very high probability and the value of ϵ will be very close to 1. In this case $m_0 = |I|$ and there is no subset with strictly better effectiveness than the database. In contrast to this scenario, if tuples of a database are highly similar, ϵ becomes small and the value of m_0 becomes very small which means a small subset of the database will deliver higher search effectiveness than the full database.

2.2 Empirical Study

The presented theoretical results in the previous section, establish 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.

2.2.1 Datasets and Query Workloads. We conduct the empirical study using two datasets from Wikipedia and StackOverflow. 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 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

[1]Available at: http://inex.mmci.uni-saarland.de/tracks/lod/2013/index.html
[2]Available at http://dumps.wikimedia.org/other/analytics

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 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.

2.2.2 Implementation. We have implemented the experiments using Apache Lucene 6.5[4] with BM25 scoring method [31]. 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. Thus, we report the results of retrieval without any boosting techniques[31].

2.2.3 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. For efficiency experiments, we do not use any multi-threading feature of the mentioned languages.

2.2.4 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.

2.2.5 Results of The Wikipedia Experiment. Figure 2 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 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.

2.2.6 Results of The StackOverflow Experiment. Figure 3 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 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.

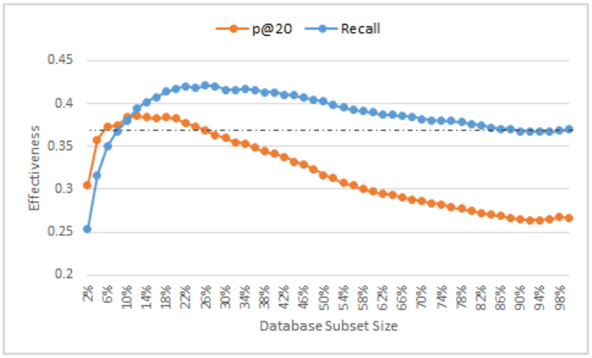

Figure 2: Effectiveness of answering INEX queries

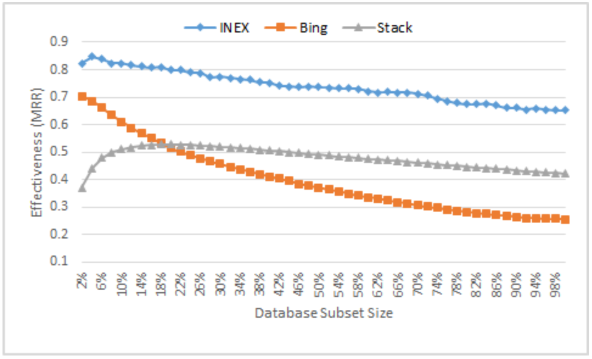

Figure 3: MRR of answering INEX, Bing and StackOverflow

[3]Available at: https://archive.org/download/stackexchange
[4]https://lucene.apache.org/

These experiments show that the theoretical results presented in Section 2.1 holds true in practice. More precisely, the results 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.

2.3 The Effective Subset of A Database

In the previous section, we showed that under certain conditions, there are subsets of a database that, on average, deliver higher search effectiveness than the full database. We call such a subset *the effective subset*. An exhaustive search of subset space to find the subset with the highest effectiveness requires exponential time computations. However, based on the results of section 2.2, one can find an approximation of the subset with the highest effectiveness using a greedy search technique. The algorithm to build an effective subset starts with an empty set and iteratively adds batches of tuples to it. The algorithm scans the tuples from most popular to list popular. After every iteration, it checks the effectiveness of answering sample queries and stops as soon as the effectiveness starts decreasing.

The effectiveness can be measured using Precision@K, recall, MRR or any other user-defined metric. By setting the effectiveness to any of these metrics, the algorithm tries to build a subset that maximizes the given metric over Q. The specified metric impacts the final size of the effective subset. For example, an effective subset for precision-at-k might not deliver a higher recall compared to the full database. As an example, for INEX experiment, the subset with 10% of the popular tuples has the best $p@20$ and the one with 22% has the best recall. Beside optimizing a subset for a single metric, it is possible to pick the subset that maximizes a metric and guarantees a minimum value for a second metric. For example, one may want to pick a subset that has the highest $p@20$ and also does not have a worse recall than the full database. The dashed line in Figure 2 specifies all the subsets that have a better recall than the full database. One may pick the best subset among these subsets to reach the highest $p@20$ while preserving the recall of the full database with similar technique explained above.

Since the size of the effective subset is usually much smaller than the full database, using this subset potentially delivers the results in a shorter time and it should be more efficient than using the full database. We will investigate the efficiency of using the effective subset in Section 5.

As mentioned in the Introduction, the effective subset may return zero relevant results for queries with unpopular relevant answers. Assume database I and a set of queries Q such that the average precision-at-k of the queries in Q over I is μ. Consider an effective subset that increases the $p@k$ of 80% of the queries by δ and decreases the $p@k$ of the rest by the same amount. Overall, the average $p@k$ will be $0.8(\mu + \delta) + 0.2(\mu - \delta) = \mu + 0.6\delta$ which is larger than its original value and is considered an improvement in the search effectiveness. However, using the subset increases the search effectiveness by sacrificing the $p@k$ of a small fraction of them. These are the queries that their relevant answers are not popular and are excluded from the subset. We name these as infrequent queries. Although infrequent queries form a smaller ratio of the whole query workload (20% in this example), a robust retrieval system should be able to handle them properly. In the next section, we present a method to addresses the issue of infrequent queries.

3 IMPROVING THE EFFECTIVENESS OF ANSWERING INFREQUENT QUERIES

In this section, we present two approaches to improve the search effectiveness of the infrequent queries. We develop two 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.

3.1 Detecting Infrequent Queries using Query Likelihood Model

Query likelihood model has been used in distributed information retrieval systems [35] to select the data source that contains more relevant answers to a given query. It measures the likelihood of a data source given a query [31]. Consider the data source I. I can be the database or any subset of it. The language model of I is defined as the multi-set of all terms that appear in I and is denoted by L. For a given query q, $P(L|q)$ denotes the likelihood of L being relevant to q. If $P(L|q)$ has a high value, it means that data source I has a higher chance in effectively answering query q. For a given L, $P(L|q)$ is computed using Bayes rule as follows:

$$P(L|q) = \frac{P(q|L)P(L)}{P(q)}$$

For a given query, its probability ($P(q)$) is independent of L and is the same for all data sources. The prior probability of a data source $P(L)$ can be computed based on different criteria. We consider a uniform prior over all data sources. Using these simplifications, one can use $P(q|L)$ to score each data source. Let query q consist of terms $q_1, \ldots, q_n$. $P(q|L) = \prod_{i=1}^{n} P(q_i|L)$ The probability of a term given a data source, $P(q_i|L)$, can be computed as the frequency of q_i in L over the size of L. If one of the terms does not appear in L, then $P(q|L)$ will be zero. To avoid zero probabilities, different smoothing techniques can be applied. We use linear interpolation as discussed in [31]. The final value of $P(q|L)$ is used as the relevance score of data source L to query q. Given the effective subset and full database with language models L_s and L_f, the source with a higher score has a better chance in effectively answering q. Thus, if $P(q|L_s) \leq P(q|L_f)$, then q is labeled as infrequent and should be submitted to the database. An experimental evaluation of this method is presented in Section 5.

3.2 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.

3.2.1 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`.

3.2.2 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.

3.2.3 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.

3.2.4 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 Section 5. Furthermore, we show the overhead of using a classifier prior to search is negligible compared to the

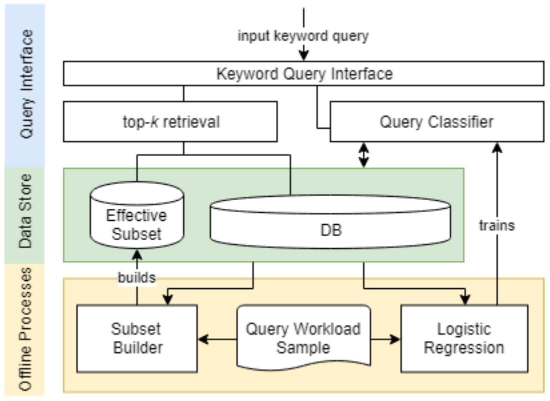

Figure 4: System Architecture

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 Section 5.

4 AN EFFECTIVE AND EFFICIENT KEYWORD QUERY SEARCH SYSTEM

In this section, we present a keyword query search system over relational databases that utilizes the effective subset of Section 2 and infrequent query detection method introduced in Section 3 to improve the effectiveness and efficiency of keyword search over large databases. Figure 4 depicts the architecture of the system with the following components:

The Off-line Processes (bottom layer) consists of two components. 1) Subset Builder finds the effective subset of the database. We explained how to build an effective subset of a single relation in Section 2.3. In Section 4.2, we present an approach to build a subset of a database with multiple relations. The output of the Subset Builder is stored in the storage layer. 2) Logistic Regression component trains the classifier that is used to detect infrequent queries. This module runs periodically to reflect the changes in users' interactions history. The trained model is stored and used by Query Classifier component.

The Data Store (middle layer) is where the system keeps the full database and its effective subset. Each database can have multiple tables. Each table has an inverted index from terms to tuples. The index also contains the statistics used to compute features for queries.

The Query Interface (top layer) is in charge of executing the query. Upon receiving a query, it uses the Query Classifier, explained in Section 3, to detect if the query is infrequent or not and submits it to the subset or full database based on this information. Next, we will explain the top-k retrieval method used in this paper.

4.1 Keyword Query Search over Relational Databases

We use the current architecture of schema-based keyword query search techniques over relational databases to retrieve the top-k answers of a given query [12]. We provide a brief overview of these techniques. We refer the interested reader to [12, 21] for more detailed explanations.

Given a keyword query, a schema-based system first selects a set of tuples (a.k.a tuple-sets) from each relation that are related to the submitted query. To find these tuple-sets and compute tuple scores, the DBMS uses an inverted index. For instance, consider a fragment of the DBLP database with relations *papers(pid, title, aid)* and *author(aid, name)*. Given query `stonebraker voltdb`, the DBMS returns a tuple-set from *papers* and a tuple-set from *authors* that match at least one term in the query. Then, it scores each tuple in the tuple sets using an IR relevance function. Next, the DBMS generates *candidate networks* of relations that are join expressions connecting tuple-sets via primary key foreign key relationship. Using each candidate network, the DBMS can join tuples from different tuple-sets to produce a single candidate result for the query. As an example, one candidate network in the mentioned example is $paper \bowtie author$. To connect the tuple-sets, a candidate network may contain base relations whose tuples may not contain any term in the query. The candidate networks are generated based on the schema of the database. For efficiency reasons, the DBMS limits the number and size of generated candidate networks. After obtaining these candidate networks, the DBMS runs many SQL queries on each of them and returns its results to the user. Each final result is a joining tree of tuples. The score of a joining tree is usually computed as the sum of scores of its tuples divided by the number of relations in the network to penalize the long joins. One of the notable examples of schema-based keyword search methods, called efficient IR-Style search, is introduced in [21]. According to [13], IRStyle Search and Cover Density Search are the most effective and efficient search techniques among schema based methods. Although Cover Density has a higher effectiveness than IRStyle, it is designed and efficient for short keyword queries. Since we aim at improving general keyword queries, we use a similar technique to IRStyle method [21] in our system.

Besides schema based systems, there is a second category of keyword search systems that are based on the data graph. These graph-based methods convert the database into a data graph. Converting a database with millions of records into a graph is memory consuming. Furthermore, how to find a meaningful sub-graph is a challenging problem [24]. For these reasons, we do not address the graph-based methods in the current paper and leave it as future work.

4.2 Building Effective Subsets over Multiple Relations

In section 2.3, we have presented an algorithm that builds the effective subset over a single table. In this section, we will extend that algorithm to handle databases with multiple tables. A naive approach is to run the algorithm on each relation R and store the subset of the relation R'. The problem with this approach is that it scans each relation independently, however, in a database with multiple relations, the answer of most of the queries is a joining tree of tuples rather than a single tuple. Thus, the subset building algorithm should take this into account. More precisely, instead of iterating over single tuples, a better approach is to iterate over joining trees of tuples. To do this, one needs the access count of

Algorithm 1: Multi Table Subset Builder

```
Data: Set of relations $\mathcal{R} = R_1 \ldots R_n$, sample queries $Q$ with
      answers, effectiveness function $e$
Result: Set of subset relations $\mathcal{S} = S_1, \ldots, S_n$ such that
        $S_i \subseteq R_i$
$P \leftarrow$ batch size, $M \leftarrow 0$
for $i$ in $(1, \ldots, n)$ do
    $S_i \leftarrow \{\}$
    Sort $R_i$ by its popularity in descending order
    Let $it_i$ be an iterator on $R_i$
end
$m \leftarrow \arg\max_i it_i.popularity()$, $t \leftarrow it_m$
while $t$ is not null do
    Scan next $P$ tuples from $it_m$ and add them to $S_m$
    $eff \leftarrow e(Q, S)$
    if $eff > M$ then
        update $M$
    else
        break;
    end
end
```

tuples and join trees. However, database systems usually store the access count of individual tuples and rarely store the access count of the join trees. The reason for this is that the number of the joins will grow exponentially as the size of the relations increases. Thus, even for databases with a moderate size, it is not feasible to store the join access count. To alleviate this problem, we use the access count of tuples participating in a join and estimate the access count of the join based on access count of the participating tuples.

Consider relations R with tuple r such that access count of r is $w(r)$. Anytime a user accesses a join that includes r, the access count of r is increased by one. This means the access count of any join tree including tuple r will be less than or equal to $w(r)$. Thus, scanning the whole database ordered by access count of tuples is an approximation of scanning the database based on access count of join trees and tuples. Based on this heuristic, we propose Algorithm 1 to build the effective subsets of a database with more than one relation. This algorithm takes a set of relations, a sample of query workload and an effectiveness metric as the input and builds the effective subset of each relation.

To build the final subset database, one can run this algorithm on tables with text attributes or the attributes that will be searched and use them to build the subset of the relation tables. As an example, consider `paper(pid, title)`, `paper-author(pid,aid)` and `author(aid, name)`. In this case, tables `paper` and `author` will be the input of the algorithm. Once their subsets are built, the subset of relation table `paper-author` is computed as all the tuples that join tuples in the subset of `paper` to their corresponding tuples in the `author` table. The processes of building the subset can be repeated periodically to reflect the changes in users' interactions and tastes over time. The output of this step is stored in the Effective Subset database in Figure 4. In the next section, we evaluate the effectiveness of the subsets built by Subset Estimator and compare it with the full database.

Table 1: Dataset Information

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

Table 2: Evaluating the built subset against full database

Experiment	Effectiveness		Time (s)	
	Subset	DB	Subset	DB
INEX-p@20	0.33	0.22	0.70	1.20
INEX-*rec*	0.29	0.22	0.70	1.20
Bing	0.51	0.08	0.37	12.80
StackOverflow	0.51	0.38	0.41	6.63

5 EXPERIMENTS

In this section, first, we evaluate the effectiveness of the subsets that are built using Algorithm 1 presented in Section 4.2. Then, we evaluate the effectiveness and efficiency of query answering using our system. Furthermore, we evaluate the accuracy of the infrequent query detection method presented in Section 3.

5.1 Experiment Setting

We use the normalized forms of Wikipedia and StackOverflow databases introduced in Section 2.2. The details of these datasets are shown in Table 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 2.2.

We use IRStyle method mentioned in Section 4 over the full database as the baseline. To create the tuple sets with relevance score we use Apache Lucene and BM25 scoring technique [31]. 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 2.2.

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. Then, we run Algorithm 1 using training queries on the given database and build a subset of its tables. 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 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 Section 2, 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 Section 2.

The second evaluation criteria for our system is the efficiency (running time) of the system. As it is shown in Table 2, the running time of the queries on the subset is much shorter than the full database. There are two major reasons for this: 1) The text index on the subset is smaller than the database, thus, looking up the keywords and creating the tuple sets takes less time on the subset compared to the database; 2) The size of the tuple sets are smaller for the subset. Thus, IRStyle Search spends less time querying these sets and submits less join queries. As it is shown in Table 2, StackOverflow queries take longer than the other queries because these queries contain 8.6 keywords per query on average and are longer than the other two query workloads. For the recall experiment (INEX-*rec*), we only measure the system's response time to retrieve top 20 results as most systems do not show all the possible results at the first run. That is why INEX-p@20 and INEX-*rec* experiments have the same running times.

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
- Database: Using the database to answer all queries
- QL: Using the query likelihood model to predict infrequent queries and reroute them to the database
- 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.

Table 3: Results of answering Bing Queries

Experiment	MRR			Time(s)
	Popular	Infrequent	All	
Subset	0.53	0.03	0.51	0.37
Database	0.07	0.51	0.08	12.80
QL	0.48	0.22	0.47	2.23
ML	0.48	0.28	0.50	2.23
Best	0.53	0.51	0.53	6.50

Table 4: Results of answering StackOverflow queries

Experiment	MRR			Time(s)
	Popular	Infrequent	All	
Subset	0.56	0.01	0.51	0.41
Database	0.36	0.50	0.38	6.63
QL	0.50	0.22	0.48	1.77
ML	0.55	0.29	0.49	1.79
Best	0.56	0.50	0.55	3.81

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 Section 3. 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 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 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 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.

In the last experiment, we evaluate our system against INEX queries. We carry out the experiment once for maximizing P@20 and once for recall. The results of this experiment are presented in Tables 5 and 6. These results follow the same trend as the previous two experiments. INEX query workload has only 145 queries compared to 6000 Bing queries and 1000000 StackOverflow queries. Because of the low number of queries, in this case, the machine learning method can not learn a very accurate model. Thus, it can not outperform the query likelihood method. These results show that, if a database system originally does not have a query workload, our system can be used with only QL infrequent query detection method and once enough queries have been logged, the

Table 5: P@20 of INEX Queries

Experiment	P@20			Time(s)
	Popular	Infrequent	All	
Subset	0.44	0.11	0.33	0.70
Database	0.17	0.31	0.22	1.20
QL	0.36	0.15	0.29	0.88
ML	0.32	0.25	0.29	0.90
Best	0.44	0.31	0.40	0.90

Table 6: Recall of INEX Queries

Experiment	Recall			Time(s)
	Popular	Infrequent	All	
Subset	0.30	0.21	0.29	0.70
Database	0.21	0.30	0.22	1.20
QL	0.29	0.21	0.29	0.73
ML	0.28	0.22	0.28	0.79
Best	0.30	0.30	0.30	0.85

system can be switched to ML mode which will deliver even higher effectiveness than the QL method.

6 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, 16, 19, 23]. Schema based approaches consider the schema as a graph and directly search the relational database by generating and executing SQL queries [20, 21, 27, 30]. 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 efficiency and 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 [32]. Our approach is different than pruning in that its objective is to increases the search effectiveness and efficiency 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 [32]. 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 [3, 15, 26, 29]. 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. [25] 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 [28]. 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.

7 CONCLUSION

The objective of this paper 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 and efficiency of the keyword search systems. In the future, we would like to expand our framework beyond keyword queries to other types of imprecise queries and support dynamic changes in the users' interaction history.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Vagelis Hristidis for providing his Java code for IRStyle keyword search method.

REFERENCES

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. 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)*. ACM, New York, NY, USA, 29–42. DOI: http://dx.doi.org/10.1145/2465351.2465355

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

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

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

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

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

[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. 2014. Overview of INEX 2014. In *Information Access Evaluation. Multilinguality, Multimodality, and Interaction*, Evangelos Kanoulas, Mihai Lupu, Paul Clough, Mark Sanderson, Mark Hall, Allan Hanbury, and Elaine Toms (Eds.). 212–228.

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] Vagelis Hristidis, Yannis Papakonstantinou, and Luis Gravano. 2003. -Efficient IR-Style Keyword Search over Relational Databases. In *PVLDB*. Elsevier, 850–861.

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

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

[24] Mehdi Kargar and Aijun An. 2011. Keyword search in graphs: Finding r-cliques. *Proceedings of the VLDB Endowment* 4, 10 (2011), 681–692.

[25] Martin L. Kersten and Lefteris Sidirourgos. 2017. 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*. http://cidrdb.org/cidr2017/papers/p58-kersten-cidr17.pdf

[26] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. 2013. Oracle TimesTen: An In-Memory Database for Enterprise Applications. *IEEE Data Eng. Bull.* 36, 2 (2013), 6–13.

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

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

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

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

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

[32] Alexandros Ntoulas and Junghoo Cho. 2007. 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*. ACM, 191–198.

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

[34] Gerard Salton and Michael J. McGill. 1986. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA.

[35] Luo Si, Rong Jin, Jamie Callan, and Paul Ogilvie. 2002. A language modeling framework for resource selection and results merging. In *CIKM*. ACM, 391–397.

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