AN ABSTRACT OF THE DISSERTATION OF

Vahid Ghadakchi for the degree of Doctor of Philosophy in Computer Science presented on December 10, 2019.

Title: Redesigning the Structure and Access Paths of Databases for Effective and Efficient Query Processing

Abstract approved: _____

Arash Termehchy

Many database users are not familiar with formal query languages, the concept of schema, or the exact content of their database. Thus, it is challenging for these users to formulate their information needs over semi-structured and structured databases. To address this problem, researchers have proposed usable query interfaces over which users can formulate their information needs without knowing about formal query languages, schema or the exact content of the database. Although the mentioned interfaces increase the usability of the databases, they inherently suffer from low effectiveness and efficiency. The recent growth in databases' content size and schema complexity only exacerbates this problem. In this dissertation, we present a set of approaches to redesign the components of database management systems to improve the effectiveness and efficiency of query processing. We present theoretical and empirical results on the impact of database size and schema complexity on the effectiveness of keyword query search. Based on these results, we propose a system that answers keyword queries more effectively. Furthermore, we present an online learning method that improves the response time of query processing over large databases.

Redesigning the Structure and Access Paths of Databases for Effective and Efficient Query Processing

by

Vahid Ghadakchi

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented December 10, 2019
Commencement June 2020

Doctor of Philosophy dissertation of Vahid Ghadakchi presented on December 10, 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 dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Vahid Ghadakchi, Author

ACKNOWLEDGEMENTS

I would like to thank my major advisor, Arash Termehchy. Arash was very patient and motivating throughout my PhD. He always encouraged me to work on fundamental and transformative research projects. He provided me the freedom to explore and work on the topics that were the most exciting for me while guiding me to stay on track towards accomplishing my PhD.

It would not be possible for me to accomplish this dissertation without the help of my committee members, Alan Fern, Prasad Tadepalli, Liang Huang and Yelda Turkan for generously dedicating their time to guide me. Their insightful comments and ideas helped develop this work. I would like to thank my collaborators Abtin Khodadadi and Mian Xie. Their hard work and ideas helped the development of the projects that are parts of this dissertation.

During my time at graduate school, I was fortunate to work with many bright people. I would like to thank students of Idea Lab, in particular Jose Picado from whom I learned a lot. I would also like to acknowledge OSU's Engineering IT support team, in particular Leanne Lai.

Finally, I would like to thank my family who provided their unconditional love throughout my graduate studies. Thank you for all your support.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

Figure | Page

LIST OF TABLES

LIST OF ALGORITHMS

Chapter 1: Introduction

Many database users, such as scientists, are *not* familiar with formal query languages and concept of database schema [72]. Thus, it is challenging for these users to formulate their information needs over semi-structured and structured databases. To address this problem, researchers have proposed usable query interfaces [72, 49, 78]. A popular class of such interfaces is keyword query interfaces (KQIs). Using a KQI, a user can express a query as a set of keywords without the need to know any formal query language or the schema of the database [66, 21, 31]. As an example, consider the DBLP (*dblp.uni-trier.de*) database which contains information on computer science publications. A fragment of this database is 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 following keyword query:

q_1 : `cluster data processing sanjay`

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 [91, 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, after submitting the keyword query q_1, the database may return the following ranked results:

```
Graph data processing on clusters, Sanjay Rakesh (2014)
```

Figure 1.1: A Fragment of the DBLP Database

ID	**Title**	**Author**	**Year**
01	MapReduce: 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
⋮	⋮	⋮	⋮

`Secure data processing in clusters, Sanjay Balraj (2015)`
`MapReduce: data processing on large clusters, Jeff Dean, Sanjay Ghemawat (2008)`
`MapReduce: a flexible data processing tool, Jeff Dean, Sanjay Ghemawat (2010)`

All these records contain the keywords in q_1. However, only the last two, i.e., papers with IDs 01 and 03, are relevant answers to the input query. Current KQIs often return too many non-relevant answers and suffer from low ranking quality (aka low search effectiveness) [4, 17, 18, 35, 91]. 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 [4, 17, 18]. Similar results have been reported in empirical evaluation of the KQIs over relational databases [35]. For example, in many cases, only 10%-20% of the returned answers are relevant to the input query [4, 17, 35].

Moreover, as KQIs have to examine numerous possible matches and answers to the input keyword query, it takes a long time for them to answer users' queries [35, 14]. The query processing time is particularly time-consuming over relational databases [14]. For queries over relational databases, a KQI has to find the relevant tuples in the base relations, score them and then compute all the 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 [14]. Since keyword queries may often be used in an interactive fashion to explore the database, users need a significantly shorter response time [31, 2].

Another class of usable query interfaces is form-based query interfaces in which users are provided with query forms. Each query form consists of fields that represent attributes of the database schema. Users provide some values for these attributes and the query interface uses the provided values to build a query, e.g., SQL expression. Then, the query interface executes this query over the database and returns its results to the users [73, 106]. If the values provided by the users does not exactly represent users' information need, the query interface may not return accurate results and may have low search effectiveness similar to the keyword query interfaces example explained above.

In the last decade, databases have grown in content size in an unprecedented rate and have become more heterogeneous. Thus, fewer users are able to precisely specify their queries using the aforementioned interfaces. Users access myriad databases mostly in domains that they are not knowledgeable. Many databases are created by integrating

information from multiple data sources. These databases often contain inconsistencies. For example, they may contain multiple different names that refer to the same entity. Even a domain expert who is aware of the content of the database, may find it difficult to formulate queries that effectively return their desired answers. In this proposal, we study the impact of database content size and schema complexity on search effectiveness. We identify the problems of state-of-the-art query interfaces and propose methods to overcome these problems.

In Chapter 2, we provide theoretical and empirical results on the impact of database size on effectiveness of answering keyword queries [54]. 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 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 effectiveness and efficiency of answering queries [55].

In Chapter 3, we provide a framework to study the impact of different schema designs on the effectiveness of answering queries. We correspond two different schemas of the same database using schema transformation rules. We present results that show transformation with specific properties decrease the effectiveness of answering queries. Using these results, one is able to compare different schemas in terms of their ability in effectively answering queries. Our investigation rejects the intuitively appealing heuristics that a schema with more relations may improve the search effectiveness more than a schema with fewer relations.

In Chapter 4, we describe the problem of processing join queries over large databases. We give an overview of the previous approaches that aim to improve the response time of join queries. Then, we introduce a novel join processing algorithm that is based on many-armed bandit algorithms and present the empirical results of using our method in comparison to the state-of-the-art.

Finally, in Chapter 5 we conclude the dissertation and recommend future research directions.

Chapter 2: Answering Keyword Queries using Effective Database Subsets

2.1 Introduction

Keyword query interfaces, despite their usability, suffer from low effectiveness in answering queries. They may return many non-relevant answers or do *not* return sufficiently enough relevant answers to the input queries. As the content size of the databases grow larger, this problem only gets worse [91].

It has been long established that in most information systems, query frequencies and their relevant answers follow a power law distribution [104, 126]. 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 [91]. 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.

Consider the DBLP database from Chapter 1 This database is shown in Figure 2.1. 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. One may run q_1 : "cluster data processing sanjay" over only these records and get a the ranked list of papers with IDs 05, 01, and 03. This result contains more relevant answer than the returned list of tuples over the entire database which would return papers with IDs 04, 05, 01 and 03. As a matter of fact, our empirical results over several real-world query workloads confirms this key intuition.

The first challenge in implementing the mentioned idea is to find such an *effective subset* of the database. If the subset contains too few tuples, it will not contain the relevant answers of majority of the queries or it may contain only a small fraction of the

Figure 2.1: A Fragment of the DBLP Database

ID	Title	Author	Year
01	MapReduce: 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
⋮	⋮	⋮	⋮

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 to answer these queries.

In this chapter, 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 in average. The improvement in 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 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 algorithm 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 (Section 2.3). Our empirical study uses several real-world datasets and query logs to evaluate the impact of using an ef-

fective subset to answering keyword queries effectively. Our results indicate that there are effective subsets in a dataset 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.

- We show how a KQI can utilize users' past interactions with the data to build the aforementioned effective subsets (Section 2.3.3).
- 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 2.4).
- We discuss and address the challenges of using our approach over relational data and address them (Section 2.5).
- 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 ranking quality, 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 2.6).

2.2 Related Works

2.2.1 Keyword Search

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 [21, 74, 45, 63]. Schema based approaches consider the schema as a graph and directly search the relational database by generating and executing SQL queries [66, 67, 87, 90]. We refer the reader to [31] 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 [34, 35]. Our proposed approach can be coupled with these search methods to increase the efficiency and effectiveness of search over large databases.

In [14], 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 explore the rest. Our approach is different because we aim to answer the queries in one shot without the need for further interactions.

Information retrieval and keyword search on unstructured and semi-structured datasets suffer from low search effectiveness in certain domains [4, 61, 58]. Our proposed framework can be applied to such retrieval systems as well to increase the search effectiveness and efficiency.

2.2.2 Index Pruning

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 score are pruned [101]. Our approach is different from pruning in that its objective is to increase 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 [6]. 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, a 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 [101]. In contrast, our system only uses one source and it does not combine the results of queries from different tiers/sources.

2.2.3 Database Caching

Caching techniques have been used in search engines [13, 24], database managements systems and multi-tier client-server web-based applications [38, 89, 5, 82]. 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 system. 3) A larger cache has a better overall performance but a larger subset does not always perform better than a smaller one.

Database caching can be done on different granularity levels: 1) table level; 2) subset of table; 3) intermediate query result and final query result level. Dar et al. have proposed one of the earliest strategies for client side database caching [38]. Their system keeps track of cached parts of tables so for each coming query it can utilize the cache as well as the database to answer the remainder of the query. Their system exploits temporal and spatial locality of the data in database. The downside of this work is that it is designed to support range queries and it does not handle joins. Lou et al. has proposed DBCache [89, 5] which addresses the earlier problems. In their work, they have developed a table level caching mechanism that uses query logs to decide which tables to cache. Also, their system uses a distributed query processing method where query plans at the cache can involve the database with a certain cost. There are other works in database caching field such as Oracle's Times Ten [82] but to the best of our knowledge, our method is the first method that improves the effectiveness of query answering as well as its efficiency.

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 database semi-autonomously remove some of its data. Kersten et al. [77] has 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 from ours as they are focused on numerical data and do not intend to increase the accuracy of answering the queries. They consider an upper bound for their database size and simulate the results of applying batch updates

and queries to the database. Their simulation uses a single table database with integer values. In their problem, they have a fixed database size and use range and aggregate queries to study the effect of discarding tuples from the database. However, in our case we do not have a tight budget on database size, rather we are optimizing the effectiveness of answering queries by changing the database size. Furthermore, our goal is to help keyword query search as opposed to range and aggregate queries on numerical data.

Machine learning based ranking methods (a.k.a learn to rank methods) use prior probabilities as a feature to train their ranking models [88]. 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.

2.2.4 Collection Size

Hawkin et al. [62] 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, 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. Despite these differences, their second hypothesis about "document frequency" feature conforms to our results when the query has a worse MRR on the full database.

Dong et al. [46] has studied the problem of picking a subset of data sources to optimize the 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.

Measuring the ability of each database to satisfy a query is a well studied area in distributed information retrieval systems. In [111], the authors use language model of the database to assign a score to each database and combine their returned results based on this score. They estimates the likelihood of generating a given keyword query using the language model of a database. First, a probabilistic language model is built for the database. This model is based on the frequency of the terms in the database. Then, the model is used to compute the probability of generating keywords of a given query. The final query likelihood score is an aggregate of the likelihood of generating each keyword in the query. While this approach is successful in IR setting, it can not directly be applied to our problem for following reason. In distributed information retrieval, each data source has a different language model. However, in our case, one of the data sources (subset) has a very similar language model to the other source (the full database) and the final score of these sources become very similar. Thus, it is difficult to decide which source is going to deliver a higher search effectiveness.

2.3 Impact of Database Size on Effectiveness of Search

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 2.5.2.

2.3.1 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 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, 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 2.1, keyword

queries are not 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 [109].

On one side, adding more entities to the database increases the number of the 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 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, $p(q)$ 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 over I and its expected value is defined as:

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

Access counts 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 $rel(q)$, $w(t) = Pr(t \in rel(q))$. $w(t)$ of a tuple in a database can be computed as the access counts of tuple t. We compute $w(t)$ as the access count of t divided by the sum of access counts of all tuples. In the example of Figure 1.1, each paper is a tuple and its popularity is computed as the number of the times the paper has been accessed divided by 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 [126]. 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 2.3.1.1. *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. Given a tuple t, let Q_t be a subset of Q such that $Q_t = \{q | q \in Q \wedge t \in rel(q)\}$. Also, let $\mathbb{1}$ be the indicator function such that $\mathbb{1}_{q(I)}(t)$ is one if $t \in q(I)$ and is zero otherwise.

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

Given random query q, $t \in q(I)$ is a Bernoulli random variable where $Pr(t \in q(I)) = \sum_{t:t \in rel(q)} \mathbb{1}_{q(I)}(t) / |Q_t|$.

$$\begin{aligned}\mathbb{E}[p@k(q(I(m)))] &= \frac{1}{k} \sum_t \frac{|Q_t|}{|Q|} Pr(t \in q(I)) \\ &= \frac{1}{k} \sum_t w(t) Pr(t \in q(I))\end{aligned}$$

Database system finds the relevant tuples to a query by assigning them relevance scores. Let us show the score of tuple t for query q as $score(t, q)$. The probability of tuple t being retrieved for query q shown by $Pr(t \in q(I))$ is equal to the probability that $score(t, q)$ is greater than at least $|I| - k$ non relevant tuples in I. Let $\bar{t}$ be a non-relevant tuple to q. We define $\epsilon_t = Pr(score(t) > score(\bar{t}))$. Then the probability that $score(t, q)$ is greater than at least $|I| - k$ non relevant tuples is equal to $\epsilon_t^{(|I|-k)}$.

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

Let r_t be the rank of tuple t based on its popularity. Given that the popularities follow

a power law distribution, the distribution function of the popularities will have the following general form:

$$w(t) = \frac{1}{H_\alpha}\frac{1}{r_t^\alpha}$$

where α is a real number greater than 1 and H_α is the $|I|^{th}$ 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:

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

Using the above simplification we have:

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

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 a 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:

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

Following theorem extends the results of Theorem 2.3.1.1 to the recall of answering queries over a database.

Theorem 2.3.1.2. *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:

$$\mathbb{E}[rec(q(I))] = \frac{1}{|Q|} \sum_q \frac{|q(I) \cap rel(q)|}{|rel(q)|}$$
$$= \frac{1}{|Q|} \sum_q \frac{1}{|rel(q)|} \sum_{t \in rel(q)} \mathbb{1}_{q(I)}(t)$$

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:

$$\mathbb{E}[rec(q(I))] \leq \frac{1}{|Q|} \sum_q \sum_{t \in rel(q)} \mathbb{1}_{q(I)}(t)$$
$$\leq \sum_t \frac{|Q_t|}{|Q|} \mathbb{E}[t \in q(I)] \leq \sum_t w(t) Pr(t \in q(I))$$

The rest of the proof is similar to the proof of Theorem 2.3.1.1. □

This result shows that, if a database is sufficiently large, there is a subset of the database such that the highest achievable $\mathbb{E}[rec(q(I))]$ over this subset is larger than the full database. Note that the threshold m_1, that is used in this theorem, can be different

than the threshold m_0 of Theorem 2.3.1.1. In fact, in most cases, m_1 is expected to have a larger value than m_0. We will discuss this in more details in Section 2.3.2.

The last metric we examine is the 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:

$$MRR = \sum_{q \in Q} p(q) R\text{-}Rank(q, I)$$

This metric is useful when the queries have a single relevant answer. Using a similar approach to Theorem 2.3.1.1 and 2.3.1.2, 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 2.3.1.1. For top-k results, the R-Rank can be expanded as:

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

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 2.3.1.1 and m_1 of Theorem 2.3.1.2 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 probability of one and 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 value of m_0 becomes very small which means a small subset of the database will deliver a higher search effectiveness than the full database.

2.3.2 Empirical Study

The presented theoretical results in 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 dataset and query logs.

2.3.2.1 Datasets and Query Workloads

We conduct the empirical study using three datasets from Wikipedia, StackOverflow and MSLR. The Wikipedia dataset contains the information on 11.2 million Wiki-pedia articles[1]. Each article has a title and a body field. This dataset also contains users' access counts for each article. The access counts are 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 [18]. It is formed of 150 keyword queries and their relevant answers over Wikipedia. For each query, 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 counts 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 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 one million queries and one million relevant answers. Furthermore, each post in StackOverflow has a view count that is the number of times a post has been

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

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.

The MSLR dataset contains 30000 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 [105].

2.3.2.2 Implementation

We have implemented the experiments using Apache Lucene 6.5[4] with BM25 scoring method [91]. For 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 counts 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[91]. For the MSLR dataset, we use the LambdaMart algorithm of pyltr library[5] to train a learn to rank model over the MSLR dataset. The dataset is partitioned to 5 folds. We use one fold to train the ranking model and use the rest of the dataset as test queries. For each test query, we obtain the ranking by applying the learned model and measure the effectiveness metrics on it.

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

[5] https://github.com/jma127/pyltr

2.3.2.3 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.

2.3.2.4 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.

2.3.2.5 Results of Wikipedia Experiment

Figure 2.2 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 does 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 2.3 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

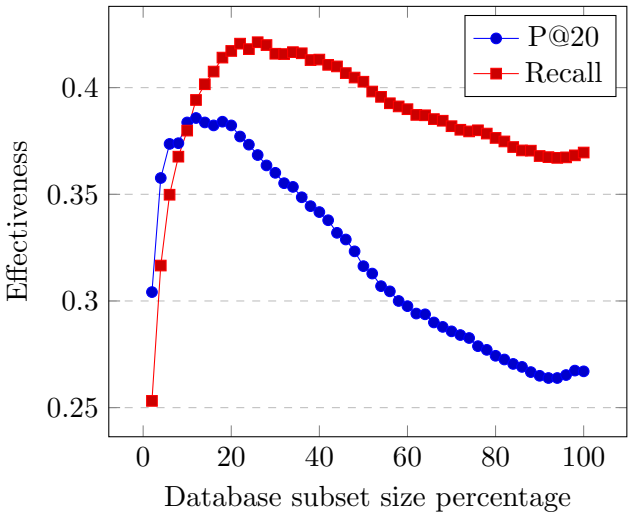


Figure 2.2: Effectiveness of answering INEX queries

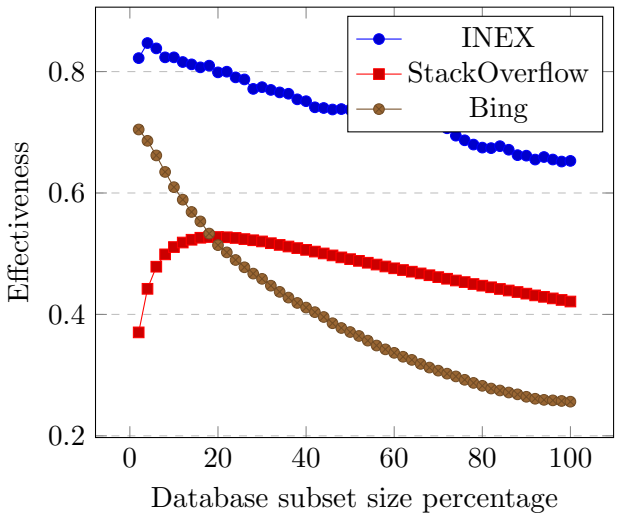


Figure 2.3: MRR of answering INEX, Bing and StackOverflow

the results to measure the effectiveness of search. For this query workload, I_2 has the highest MRR and for subsets larger than I_2, MRR has a decreasing trend.

2.3.2.6 Results of StackOverflow Experiment

Figure 2.3 shows the effectiveness of query answering over different subsets of 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.

2.3.2.7 MSLR

The MSLR dataset includes the BM25 score of each query-url pair. We use this score to simulate the retrieval process for subsets with different sizes. Figure 2.3 shows the effectiveness of answering the queries over different subsets of this dataset. In this experiment, MRR has a similar trend to previous experiments. Since results have different relevancy scores, we also measure the NDCG of answering queries which is shown in the figure. As it is shown, NDCG has a more significant drop compared to MRR. This shows that tuples with higher popularity have higher relevance score. Thus, for retrieval systems with non-binary relevance scoring, the impact of subset size is more important. In this experiment, we don't measure the recall because each query in MSLR dataset has 100 URLs on average and thus it has less than 100 relevant answer and a system that returns more than 100 results will have perfect recall.

These experiments show that the theoretical results presented in Section 2.3.1 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 the effectiveness becomes more significant. In next section, we use these results to build a subset of the database that delivers a significantly higher effectiveness in answering queries.

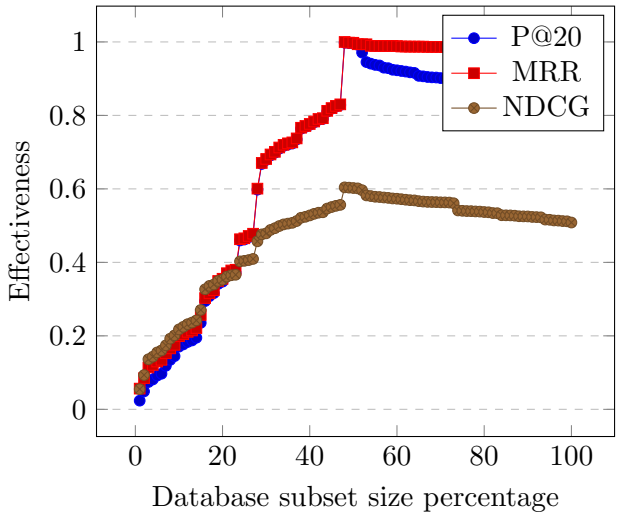


Figure 2.4: Impact of subset size on MSLR queries

2.3.3 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*. Exhaustive search of subset space to find the subset with the highest effectiveness requires exponential time computations. However, based on the results of section 2.3.2, one can find an approximation of the subset with the highest effectiveness using a greedy search technique. The algorithm to build the 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 2.2 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 2.6.

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.

2.4 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 delivers 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.

2.4.1 Detecting Infrequent Queries using Query Likelihood Model

Query likelihood model has been used in distributed IR systems [111] 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 [91]. Consider 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 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 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 [91]. 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. The results of using this method is presented in Section 2.6.

2.4.2 Detecting Infrequent Queries using Machine Learning

In this section, we present a method to train 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.

2.4.2.1 Content Based Features

Content based features are based on 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, 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`.

2.4.2.2 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 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 `mapreduce framework`. The term `framework` can happen in tuples with different popularities thus it's 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.

2.4.2.3 Query Difficulty Based Features

IR researchers have developed query difficulty metrics to predict the quality of the search results of a query [27]. 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 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 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 [27]. 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 subset or full database based on the results of the classifier.

2.4.2.4 Training The Infrequent Query Classifier

We use logistic regression method to train our classifier. Logistic regression is a good fit for this problem because of the following reasons. First, it has a higher interpretability and it is easier to see which features have 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 full database is larger than the subset, we label the query as infrequent. Otherwise, it is labeled as 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 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 search and submit the infrequent queries to the full database. We evaluate the effectiveness of this system in Section 2.6. 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 Section 2.6.

2.5 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 2.3 and infrequent query detection method

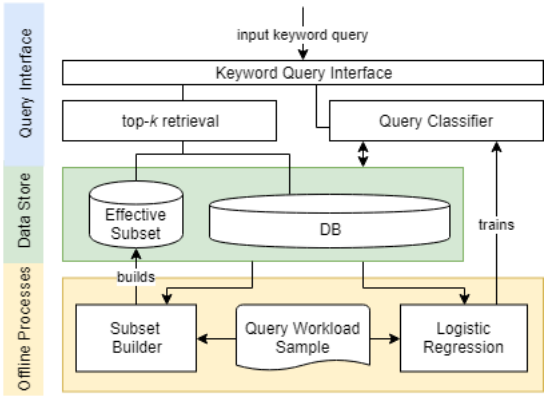


Figure 2.5: System Architecture

introduced in Section 2.4 to improve the effectiveness and efficiency of keyword search over large databases. Figure 2.5 depicts the architecture of the system with 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 2.3.3. In Section 2.5.2, we present an approach to build a subset of a database with multiple relations. The output of 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 2.4, 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 chapter.

2.5.1 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 [31]. We provide a brief overview of these techniques. We refer the interested reader to [67, 31] 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 return 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 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 [67]. According to [34], 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 [67] 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 [76]. For these reasons, we do not address the graph-based methods in the current work and leave it as a future work.

2.5.2 Building Effective Subsets over Multiple Relations

In section 2.3.3, 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, 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 counts of tuples and join trees. However, database systems usually store the access counts of individual tuples and rarely store the access counts of the join trees. The reason for this is that 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 counts. To alleviate this problem, we use the access counts of tuples participating in a join and estimate the access count of the join based on access counts 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 that, the access counts 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 counts 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 following tables: `paper(pid, title)`, `paper-author(pid,aid)` and `author(aid, name)`. In this case, tables `paper` and `author`

Algorithm 1 Multi Table Subset Builder

Data: Set of relations $\mathcal{R} = R_1 \dots R_n$, sample queries Q with answers, effectiveness function e

Result: Set of subset relations $\mathcal{S} = S_1, \dots, S_n$ such that $S_i \subseteq R_i$

$P \leftarrow$ batch size, $M \leftarrow 0$

for i *in* $(1, \dots, n)$ **do**

  $S_i \leftarrow \{\}$

  Sort R_i by its popularity in descending order

  Let it_i be an iterator on R_i

$m \leftarrow \text{argmax}_i\, it_i.popularity()$, $t \leftarrow it_m$

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;

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 `authors` 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 2.5. In next section, we evaluate the effectiveness of the subsets built by subset estimator and compare it with the full database.

2.6 Experiments

In this section, first we evaluate the effectiveness of the subsets that are built using Algorithm 1 presented in Section 2.5.2. 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 2.4.

Table 2.1: Dataset Information

Dataset	#Tuples	#Relations	Size (GB)
Wikipedia	130M	5	35
StackOverflow	304M	5	2.3
MSLR	31K	1	4.1

2.6.1 Experiment Setting

We use the normalized forms of Wikipedia and StackOverflow databases introduced in Section 2.3.2. The details of these datasets are shown in Table 2.1. 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 users' access counts for articles, images and links. The StackOverflow dataset contains the information of StackOverflow posts with following tables: *posts, post-comment, comments, post-tag, tags* and their access counts. The attributes used for keyword 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 2.3.2.

We use IRStyle method mentioned in Section 2.5 over full database as the baseline. To create the tuple sets with relevance score we use Apache Lucene and BM25 scoring technique [91]. 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 score less than $\frac{s}{2}$ from the tuple set. This helps the IRStyle method to process the queries a reasonable time. For experiment on p@20 and MRR, we retrieve the top 20 tuples and for recall we retrieve the top 100 tuples. The experiment environment is similar to Section 2.3.2.

2.6.2 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 train 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

Table 2.2: Evaluating the built subset against full database

Experiment	Effectiveness		Time (s)	
	Subset	DB	Subset	DB
INEX-*p*@20	0.33	0.22	0.7	1.2
INEX-*rec*	0.29	0.22	0.7	1.2
Bing	0.51	0.08	0.47	12.8
StackOverflow	0.48	0.34	0.41	6.63

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 2.2. The rows are associated with experiments and the columns are the results of that experiment. As shown in the table, the subset deliver a higher effectiveness than the baseline in all four experiments. The highest gain happens in the Bing experiment. This is because for Bing experiment, the effective subset is much smaller (2%) and as discussed in Section 2.3, a smaller subset results in much less search mistakes by the database system. Furthermore, the effective subset for recall has the largest size as explained in Section 2.3.

The second evaluation criteria for our system is the efficiency (running time) of the system. As it is shown in Table 2.2, the running time of the queries on subset are 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 subset. Thus, IRStyle Search spends less time querying these sets and submits less join queries. As it is shown in Table 2.2, 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 first run. That is why INEX-p@20 and INEX-*rec* experiments have the same running times.

2.6.3 Evaluating The Infrequent Query Detection

In this section, we evaluate query type prediction method. The objective of query type prediction is to detect the infrequent queries and improve their results while maintaining a 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.

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 2.4. 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 2.3. 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

Table 2.3: 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	12.8
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	6.5

Table 2.4: Results of answering StackOverflow queries

Experiment	MRR			Time(s)
	Popular	Infrequent	All	
Subset	0.63	0.01	0.55	0.41
Database	0.41	0.47	0.42	6.63
QL	0.57	0.22	0.51	1.77
ML	0.57	0.25	0.53	1.79
Best	0.63	0.48	0.60	3.81

queries, the subset outperforms all other methods. However, it has a very low MRR of 0.03 for infrequent queries. The ML methods 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 the system using StackOverflow dataset using a similar approach as above. The results of this experiment are shown in Table 2.4. 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 0.25 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 2.5 and 2.6. These results follow a 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

Table 2.5: P@20 of INEX Queries

Experiment	P@20			Time(s)
	Popular	Infrequent	All	
Subset	0.44	0.11	0.33	0.7
Database	0.17	0.31	0.22	1.2
QL	0.36	0.15	0.29	0.88
ML	0.32	0.25	0.29	0.9
Best	0.44	0.31	0.40	0.9

Table 2.6: Recall of INEX Queries

Experiment	Recall			Time(s)
	Popular	Infrequent	All	
Subset	0.30	0.21	0.29	0.7
Database	0.21	0.30	0.22	1.2
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

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

2.7 Conclusion

Objective of this chapter 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 indicate that it is successful in increasing the effectiveness and efficiency of the keyword search systems.

In the next chapter, future, we would like to expand our framework beyond keyword queries and support dynamic changes in the users' interaction history.

Chapter 3: Schema Capacity

3.1 Introduction

Many users are able to formulate correct queries as far as the query language, and schema of the database are concerned [96, 97, 98, 93, 28]. However, a lot of users do not have sufficient information about the database content. Hence, they may refer to data items using terms that are different from the ones used to represent these data items in the database. For instance, a user who is searching for the papers written by *Hector Garcia Molina* in the database fragment in Figure 3.1, may use *Molina*, *Hector*, or *Victor Molina* in his query to refer to *Hector Garcia Molina*. This phenomenon is generally called *vocabulary gap* [91]. Vocabulary gap between users and a database may also happen when the users are not familiar with the schema of the database. For instance, they may refer to the schema elements using different names from the ones used in the database or use the incorrect attributes to join relations [65, 32]. In this Chapter, we focus on vocabulary gap between the data items, i.e., domain constants, used in users' queries and the database. We call a query that suffers from the problem of vocabulary gap, an *imprecise query*.

Because the domain constants in an imprecise query and the ones in the user's desired tuples in the database do not generally match, the query interface may not return any answer or may return mostly undesired answers for the query. For example, if the user refer to *Hector Garcia Molina* in his query over the database fragment in Figure 3.1 as *Victor Molina*, he will not obtain any answer. To answer these queries, query interfaces use some *retrieval functions*, such as traditional TF-IDF formula or query expansion, to discover the most similar domain constants in the database to the ones used in the query. Then, they replace the domain constants in the query with the discovered ones and return the results of the obtained query [96, 97, 98, 99, 93, 50, 91]. The user would like to find majority of her desired answers in the returned results, i.e., achieve *high recall*, and do not see many non-relevant answers in the results, i.e., get *high precision* [91].

Publication

id	title	author	venue	type
11	Information Preserving Embedding	Wenfei Fan	TODS	Journal
12	A web of concepts	Nilesh N. Dalvi	SIGMOD	Conference
13	Views and Query Rewriting	Alan Nash	TKDE	Journal
14	A survey of schema matching	Erhard Rahm	PVLDB	Journal
15	A Semantic Search Engine for XML	Sara Cohen	VLDB	Conference
16	Extending the relational model	Edgar Frank Codd	TKDE	Journal
17	Answering queries using views	Alon Y. Halevy	PVLDB	Journal

Figure 3.1: A database instance of schema DBLP1.

Users often transform their databases and represent it under a new schema for various reasons, such as achieving interoperability, data quality, and /or performance [68, 9, 33, 16, 69, 94, 52]. Hence, a natural and important question is how transforming a database affect the precision and recall of answering imprecise queries, given the query interface uses the same retrieval function over both original and transformed databases. Because user satisfaction is of paramount importance for enterprises, they may like to avoid transformations that reduce the precision and /or recall of imprecise queries. Further, it is notoriously challenging to improve the effectiveness of retrieval functions [8, 7]. Therefore, the effectiveness of current retrieval functions have yet to meet users' expectations [120, 114, 32]. Knowing the database transformations that are likely to improve precision or recall for imprecise queries, one may *simulate* that transformation by create a set of views over the database and design query form interfaces according to this set of views.

To the best of our knowledge, there has not been any formal study on the impact of database transformations on the precision and recall of answering imprecise queries. Generally, finding these impacts are left to the intuition. In this Chapter, we provide a formal framework to investigate the effects of database transformations on effectiveness of answering queries. Using this framework, we compare different relational schemas in terms of their abilities to answer imprecise queries effectively. In particular, our investigation suggests that some intuitively appealing heuristics, such as splitting a relation to multiple relations in the database, do not generally improve and in some cases lower the values of precision and recall of answering imprecise queries of the transformed database. In this chapter, we present the following contributions:

- We provide an abstract model for answering imprecise queries over relational data-

bases. Since our model does not depend on any particular retrieval function, it can be used to reason about the effectiveness of answering imprecise queries in general case.

- We introduce and formally define the notions of recall and precision preserving transformations. The value of recall (precision) of answering an imprecise query over a source database of a recall (precision) preserving transformation is equal or greater than the recall (precision) of answering the same query over the transformed database. We use these notions to compare different relational schemas in terms of precision and recall of answering the same imprecise queries.

- We present two general class of database transformations and provide the conditions under which these transformations are not precision (recall) preserving.

- Since both high precision and high recall cannot be usually obtained for an imprecise query q, users often relax q, i.e., submit a p, where $q \subset p$ in order to achieve higher recall for their queries in the expense of possibly losing precision. We prove that a transformation whose inverse is a union of conjunctive queries with non-empty active domain, is still not recall preserving even after relaxing queries over its target database.

This rest of this chapter is organized as follows. Section 3.2 describes the related works and Section 3.3 defines some basic concepts. Section 3.4 introduces a formal framework to reason about the effectiveness of answering imprecise queries. In Section 3.5, we present our results and in Section 3.6 we evaluate our results using experimental studies.

3.2 Related Works

3.2.1 Imprecise Queries and Usable Query Interfaces:

Researchers have developed various systems and algorithms to answer imprecise queries over databases, particularly in the last decade. We refer the readers to interesting tutorials and surveys in recent database conference and journals [128, 72, 32]. We build on this line of research by providing a formal framework to analyze the impact of database transformations on the effectiveness of answering imprecise queries.

3.2.2 Schema Transformation:

One has to transform schemas in various applications, such as schema normalization [33, 16, 9], data integration [94, 95], model translation [15, 43], data exchange [51], and query optimization. Some of these applications require that the information contained in the original schema can be reconstructed from the transformed schema and every query over the original schema can be translated to an equivalent query over the transformed one[9, 68, 69, 52, 94, 3]. The first property, which is called *invertibility* [52] (or *dominance* [68]), is generally enforced by restricting the schema transformations to invertible functions. The second property, which is called *query preservation* [52] (or *query dominance* [68]), coincides with invertibility provided that the query language is sufficiently expressive [52]. We build on this line of research by investigating how transforming a schema affects its ability to effectively answer vague queries. The transformations in our work does not necessarily need to have an inverse.

Using a similar technique to data integration methods [94, 95], we describe the transformed schema using a set of view definitions. There are different approaches to answer a query over a set of views [60]. We use the inverse rule method because of its simplicity and modularity.

3.3 Preliminaries

Let *Attr* be a countably infinite set of symbols that contains the names of *attributes*. The *domain* of attribute A is a countably infinite set of values that A may contain. Most usable query interfaces consider the domain of all attributes in a schema to be text values [65, 36, 99, 97]. Thus, we assume that the all attributes share the same domain *dom*. Each element in *dom* is a *constant*. Let *relname* be a countably infinite set of text values of *relation* names. A *relation* R is defined by a name and is associated with a fixed set of attributes denoted as $sort(R)$. We show relation R as $R[U]$, where $U = sort(R)$. A database schema is a tuple $\mathcal{S} = \langle \mathcal{R}, \Sigma \rangle$ where $\mathcal{R}$ is a nonempty set of relations R and Σ is a set of constraints [1, 9]. A constraint makes restrictions on properties of data that might be stored in the database. A domain constraint restricts the domain of an attribute to a nonempty subset of *dom*. A *tuple* t over relation $R[U]$ is a total map from U to *dom*.

The relation instance I_R of relation $R[U]$ is a finite set of tuples with sort U. A database instance of database schema $\mathcal{S}$ is a mapping $I_\mathcal{S}$ over $\mathcal{S}$ that associates each relation $R \in \mathcal{S}$ to a relation instance I_R. Set of all instances of $\mathcal{S}$ is shown as $I(\mathcal{S})$. The *active domain* of a relation instance I_R, shown as $adom(I_R)$, is the set of all constants in *dom* that appear in I_R. We define and denote the active domains for a database instance similarly. Also, $adom(A, I_R)$ denotes the set of all constants in *dom* that are assigned to attribute A in the tuples of relation instance I. We drop I_R in $adom(A, I_\mathcal{S})$ when it is obvious from the context.

Let *var* be an infinite set of variables that range over the elements in *dom*. Each element in the set of $var \cup dom$ is called a *term*. A *free tuple* over $R[U]$ is a function u from U to $var \cup dom$. We use notation $\vec{X}$ to show tuples of attributes and $\vec{u}$ to show a tuple or a free tuple. The expanded form will be: $\vec{u} = u_1, \ldots, u_m$.
Given a database schema $\mathcal{S}$, a *conjunctive query* (CQ) over $\mathcal{S}$ takes the form of:
$Q(u) :- R_1(u_1), \cdots, R_n(u_n)$ where $n > 0$, $R_j \in \mathcal{S}$, $1 \leq j \leq n$, u_j is a free tuple, $Q \notin \mathcal{S}$, and u is a function from $sort(Q)$ to *var*. Each variable that occurs in u must also appear at least once in a u_j. We call $R_j(u_j)$ a predicate and S the head of q. A *union of conjunctive queries* (UCQ) over $\mathcal{S}$ is a finite set of CQs, i.e. its rules, that share the same head which is a relation. Extending UCQs with inequality operator is shown as $UCQ^{\neq}$. We limit our investigations to $UCQ^{\neq}$ because this family covers many queries that particularly have the issue of vocabulary gap. For instance, most of the form-based query interfaces translates user inputs to UCQs. The *active domain* of a query q, shown as $adom(q)$, is the finite set of all domain constants that occur in q. We show results of query q over database instance $I_\mathcal{S}$ as $q(I_\mathcal{S})$. Result of q over a single relation instance I_R is denoted as $q(I_R)$. Queries q and q' over database schema $\mathcal{S}$ are considered *equivalent*, $q \equiv q'$, if and only if for every instance $I_\mathcal{S}$ of $\mathcal{S}$ we have $q(I_\mathcal{S}) = q'(I_\mathcal{S})$.

3.4 Noisy Channel Model

Throughout this chapter, we will use three schemas that are explained in the following example.

Example 3.4.0.1. *The first schema (DBLP1) is shown in Table 3.1 and has a single relation containing publications of different authors in different venues. The second schema (DBLP2) is shown in Table 3.2. This schema contains two relations: Article*

Article

id	title	author	venue
11	Information Preserving Embedding	Wenfei Fan	TODS
13	Views and Query Rewriting	Alan Nash	TKDE
14	A survey of schema matching	Erhard Rahm	PVLDB
16	Extending the relational model	E. F. Codd	TKDE
17	Answering queries using views	Alon Y. Halevy	PVLDB

Paper

id	title	author	venue
12	A web of concepts	Nilesh N. Dalvi	SIGMOD
15	A Semantic Search Engine for XML	Sara Cohen	VLDB

Figure 3.2: A database instance of schema DBLP2.

Journal

id	title	author	venue
11	Information Preserving Embedding	Wenfei Fan	TODS
13	Views and Query Rewriting	Alan Nash	TKDE
16	Extending the relational model	E. F. Codd	TKDE

VLDB

id	title	author	type
14	A survey of schema matching	Erhard Rahm	PVLDB
15	A Semantic Search Engine for XML	Sara Cohen	VLDB
17	Answering queries using views	Alon Y. Halevy	PVLDB

Conference

iid	title	author	venue
12	A web of concepts	Nilesh N. Dalvi	SIGMOD

Figure 3.3: A database instance of schema DBLP3.

and Paper. Relation Article contains the publications with a journal venue and Paper contains the publications with a conference venue. The third schema (DBLP3) has three relations: Journal, VLDB and Conference. VLDB contains the publications that are published in 'VLDB Conference' or 'PVLDB Journal'. Journal contains journal articles except those with 'PVLDB' as venue. Conference contains conference papers except those with 'VLDB' as venue. There is a domain constraint over typ attribute of relation VLDB that restricts the domain of typ to $\{VLDB, PVLDB\}$.

3.4.1 Vocabulary Gap

Users are often familiar with the schema of the database but not its content. Therefore, they may submit queries that are correctly framed under the schema of the database but

have only partial and /or inaccurate information about domain constants [73, 97, 98, 93]. Let query q express the true information need of a user and q' be the query that the user submits. In a lot of cases, user fails to formulate an exact query and consequently $q' \not\equiv q$. q' is a inaccurate/noisy version of the desired query q.

Example 3.4.1.1. *Suppose a user wants to retrieve papers by "Sarah Cohen" published in the VLDB conference. The database instance has schema DBLP2 and its fragments are shown in Table 3.2. The user's desired query, q, is:*

$A(x) :- paper(x, y, \texttt{Sarah Cohen}, \texttt{VLDB Conference})$

Nonetheless, the user may know only the last name of Sara Cohen and submit query q'_1 as follows.

$A(x) :- paper(z, y, \texttt{Cohen}, \texttt{VLDB Conference})$

It is also likely that she may vaguely recall the name of Sara Cohen as another author "Sarah Kuper" and may submit the following query q'_2:

$A(x) :- paper(x, y, \texttt{Sarah Kuper}, \texttt{VLDB Conference})$

It is also possible that the user expands the acronym "VLDB" and submits q'_3:

$A(x) :- paper(x, y, \texttt{Sarah Cohen}, \texttt{Very Large Databases})$

The phenomena that users use different terms than the ones in data sources to refer to data items is called *vocabulary gap* [91]. Vocabulary gap is one of the main challenges in satisfying users' information needs in usable query interfaces, such as keyword or form-based query interfaces [91]. For example, due to the vocabulary gap between the user's queries and the database values in Example 3.4.1.1, the user is not able to submit her desired query and query interface has to predict the desired query based on the submitted query.

Vocabulary gap happens because of different reasons:

- Under-specified Constants: The submitted query is called under-specified when a constant in it is a subset of the corresponding constant in the desired query. Query q'_1 in Example 3.4.1.1 is an under-specified version of the desired query which leads to a vocabulary gap.

- Partially Noisy Constants: We say a constant in a submitted query is partially noisy when a part of the constant is different from the corresponding constant in the desired query. The constant `Sarah Kuper` in Example 3.4.1.1 is a partially noisy representation of the constant `Sarah Cohen` in the desired query q.

- Equivalent Constants: Given a desired query with constant c, user may submit a query with constant c' where c' is a synonym, acronym, abbreviation or expansion of c. q'_3 in Example 3.4.1.1 is a sample of this case. The submitted query does not have any mistakes or noise, however, this query suffers from the vocabulary gap problem and submitting it to a traditional RDBMS with exact matching technique will result in zero relevant answers.

- Spelling Errors: This is one of the most common errors in users' queries. Consider the query:
  $A(x) : -paper(x, \texttt{treep patterns}, \texttt{Sarah Cohen}, z)$
  In this query, the word `treep` is misspelled. The correct form of this word can be either `tree` or `treap` [1]. A modern query interface has to be able to handle spelling errors properly.

Given submitted query q' with a noisy constant c', the usable query interface has to predict its corresponding desired query. We denote this predicted query by $f(q')$. In a form-based query interface, an inaccurate query q' precisely conveys the user's information need as far as schema elements and their relationships are concerned and the noise is in the constants of the query. Thus, a modern query interface uses different techniques to predict the correct constant for c'. It may use similarity measures, lexicons, query history data, etc., to form a set of predicted values ($\mathcal{PV}$) for each noisy constant. Next, query interface will build new queries by replacing the noisy constant with predicted values in $\mathcal{PV}$. The set of these queries will form $f(q')$.

Example 3.4.1.2. *Consider database fragment in Table 3.2. Assume*
$q : A(x) : -\ paper(x, y, \texttt{Sarah Cohen}, z)$
$q' : A(x) : -\ paper(x, y, \texttt{Cohen}, z)$
A possible set of predicted values for constant `Cohen` *in* q' *can be*

$$\mathcal{PV} = \{\texttt{Sarah Cohen}, \texttt{Leo Cohen}, \texttt{Cohen Corwin}\}$$

[1] Treap is a type of binary tree data structure

Thus, $f(q')$ will be:

$$A(x) : -paper(x, y, w, z),$$
$$w \in \{\texttt{Sarah Cohen}, \texttt{Leo Cohen}, \texttt{Cohen Corwin}\}$$

Assumption 3.4.1.1. *Given database relation R with attribute $A \in R$ and relation instances I_R and $J_R \supset I_R$, if $adom(A, J_R)$ is sufficiently larger than $adom(A, I_R)$, then*

$$|\mathcal{PV}(A, J_R)| > |\mathcal{PV}(A, I_R)|$$

Assume noisy query q' on database instance I. Depending on the type of the vocabulary gap in q', query interface uses different approaches to build $\mathcal{PV}$. It can utilize the database instance I or rely on other available information. Based on this criteria, we divide query interfaces into two categories.

- The first type build $\mathcal{PV}$ independent of the database instance I. As an example, consider the following query:
  $q'(x) : -paper(x, \texttt{Treep Patterns}, y, z)$
  In this query, the constant `Treep` is misspelled. The set of predicted values for this constant is $\mathcal{PV} = \{\texttt{Tree}, \texttt{Treap}\}$. The values in $\mathcal{PV}$ are predicted using a fixed lexicon, and they will be the same for any database instance I.

- The second type of query interfaces make use of I to build $\mathcal{PV}$. Consider following noisy query:
  $q'(x) : -paper(x, y, \texttt{Cohen}, z)$
  where `Cohen` is a noisy constant that binds attribute *Author*. In this case, query interface will use similarity measures such as tf-idf to find the proper set of constants from I that match the noisy query. As an example, $\mathcal{PV}$ might be:

  $$\mathcal{PV} = \{\texttt{Sarah Cohen, Leo Cohen, Cohen Kuper}\}$$

  The values in $\mathcal{PV}$ are extracted from the active domain of attribute *Author* in database instance I and changing the database instance can lead to a different $\mathcal{PV}$.

3.4.2 V-Isomorphism

Next, we formalize the relationship of predicted query $f(q')$ with submitted query q'. For the sake of simplicity, we assume that none of the rules in a query is a subset of another rule. More formally, given a query with set of rules $\{r_1, \ldots, r_n\}$, for all r_i, r_j, $1 \leq i, j \leq n$, $i \neq j$, we have $r_i \nsubseteq r_j$. If a query does not have this property, it can be automatically and efficiently converted to such a form [1].

Definition 3.4.2.1. *Given conjunctive queries q and r over schema $\mathcal{S}$, a variable isomorphism (v-isomorphism) θ from q to r is a bijection between the sets of variables and a relation between the sets of constants in q and r such that $\theta(q) = r$.*

We call two queries *v-isomorphic*, if and only if there is a v-isomorphism between them. For instance, q and q' in Example 3.4.1.1 and the set of queries in Example 3.4.1.2 are v-isomorphic. The above definition can be extended to represent the relationship between a conjunctive query and a UCQ.

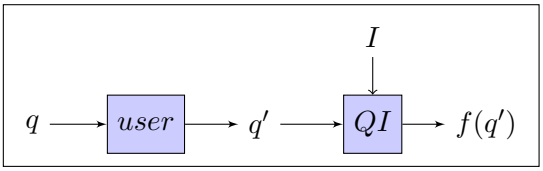
Definition 3.4.2.2. *Given UCQs p and q over schema $\mathcal{S}$, we say there is a v-isomorphim from p to q iff there is a v-isomorphism between each rule in p and a rule in q.*

Note that the above definition is not symmetric. There can be rules in q that doesn't have v-isomorphic counterparts in p. Depending on the submitted query and the content of the database, query interface may replace a submitted rule with multiple rules. Each new rule will be v-isomorphic to the original rule.

3.4.3 Users and Query Interface as Noisy Channels

One can think of a user as a noisy channel such that the input of this channel is user's information need and the output is user's submitted query. The input and output of this channel are v-isomorphic. In other words this channel preserves the basic schema of the query, but may modify its constants or add or remove some rules. In a similar fashion, query interface or QI for short can be considered as a noisy channel such that its input is user's submitted query q' and its output is the predicted query $f(q')$. The input and output of this noisy channel are also v-isomorphic to the output. Figure 3.4 shows the relation between these channels and database instance I. As it is shown in Example

Figure 3.4: Vocabulary gap noisy channel model



3.4.1.2, query interface uses the active domain of the attribute to predict queries. The input from I to QI in the Figure 3.4 shows this fact.

Given a desired query q and a noisy version of it q', submitted by the user, the effectiveness of answering q' over a database instance is generally measured in terms of *recall* and *precision* [91].

Definition 3.4.3.1. *Given database instance I and submitted query q', let q and $f(q')$ be the desired and predicted queries. The* **precision** *and* **recall** *of the result of $q'(I)$ are defined as follows:*

$$pre(q', I|q) = \frac{|f(q')(I) \cap q(I)|}{|f(q')(I)|}.\ rec(q', I|q) = \frac{|f(q')(I) \cap q(I)|}{|q(I)|}$$

Note that $f(q')$ is the predicted query by the query interface. We drop q from the right hand side of the definition when it is obvious from the context. The precision of a query result measures the portion of the true positive answers in the result. The recall of the result quantifies its coverage. Ideally, one would like to get answers with both high precision and recall. However, it is generally hard to maximize both these metrics at the same time. Thus, depending on the type of user's information need, one may like to sacrifice one of them in order to maximize the other. For instance, a medical researcher who searches the literature for genes that correlate with a certain disease would like the query interface to cover all possible answers and may not mind checking some false positives in return. On the other hand, a traveler who looks for nearby restaurants may sacrifice the coverage of a query to find some reasonably close restaurants. The aforementioned trade-off between precision and recall is an important tool for satisfying users' information needs [91].

3.5 Schema Transformation And Effectiveness of Answering Queries

Database administrators transform database schemas for various reasons such as schema normalization, data integration, and query performance improvement. Given a (source) schema $\mathcal{S}$, a schema transformation can be represented with a set of view definitions $\mathcal{V} = \{v_1, \ldots, v_m\}$, where each v_i, $1 \leq i \leq m$ is a query over $\mathcal{S}$. For each v_i we denote its head and body as $Head(v_i)$ and $Body(v_i)$ respectively. We express each view definition in $UCQ^{\neq}$. Consider DBLP1 from Table 3.1 that contains a publications table and DBLP2 from Table 3.2 that contains a table for conference papers and another table for journal articles. One can define DBLP2 as a set of views over DBLP1 as follows:

$$Article(x_1, x_2, x_3) : -Publication(x_1, x_2, x_3, x_4, Journal)$$
$$Paper(y_1, y_2, y_3) : -Publication(y_1, y_2, y_3, y_4, Conference)$$

For each instance $I \in I(\mathcal{S})$, $\mathcal{V}(I)$ is the database instance obtained by applying the view definitions in $\mathcal{V}$ to I. Given query q over $\mathcal{S}$ and q_v over $\mathcal{V}$, q_v is called an exact rewriting of q over $\mathcal{V}$ if for every $I \in I(\mathcal{S})$ and $J = \mathcal{V}(I)$ we have $q_v(J) = q(I)$. Continuing our example, assume the user wants to retrieve conference papers of Sarah Cohen from databases of Table 3.1 and 3.2. In this case, the query over DBLP1 and its exact rewriting over $\mathcal{V}$ are:

$$q(x_1) : -publications(x_0, x_1, Sarah\,Cohen, x_2, Conference)$$
$$q_v(x_1) : -paper(x_0, x_1, Sarah\,Cohen, x_2)$$

To obtain q_v from query q and view definitions $\mathcal{V}$, we use the inverse rule algorithm introduced in [47]. The exact rewriting of a query does not always exist. We extend our results to cover such cases as well.

Definition 3.5.0.1. *Given schema $\mathcal{S}$ and its transformation $\mathcal{V}$, we say the transformation is precision preserving iff for every query q over $\mathcal{S}$, database instance $I \in I(\mathcal{S})$ and its transformation $J = \mathcal{V}(I)$, there is an exact rewriting q_v over $\mathcal{V}$ such that $pre(q(I)) \leq pre(q_v(J))$.*

A recall preserving transformation is defined similarly. In the rest of this section, we study the effectiveness of answering queries over a schema and its transformations.

Next theorem formalizes the first case that using views reduces the effectiveness of query answering:

Theorem 3.5.0.1. *Given schema $\mathcal{S}$ and a set of view definitions $\mathcal{V}$, $\mathcal{V}$ is not precision preserving if there is a disjunctive view $V \in \mathcal{V}$ such that:*

$$adom(Body(V)) \subsetneq adom(Head(V))$$

Proof. Given relations $R_1(A_1, B_1, C_1), \ldots, R_n(A_n, B_n, C_n)$, assume $V(A, B, C, D)$ has the following general form:

$$\begin{aligned} V(x, y, c_1, d_1) &:- R_1(x, y, c_1) \\ V(x, y, c_2, d_2) &:- R_2(x, y, c_2) \\ &\vdots \\ V(x, y, c_n, d_n) &:- R_n(x, y, c_n) \end{aligned}$$

Consider desired query $q(x) : -R_1(x, b, c_1)$ and its rewriting $q_v(x, y) : -V(x, b, c_1, d_1)$. Let us denote the noisy versions of these queries as $q'(x) : -R_1(x, b', c'_1)$ and $q'_v(x, y) : -V(x, b', c'_1, d'_1)$. Assume $I \in \mathcal{J}(\mathcal{R})$. We have $q(I) = q_v(I)$. Assuming that $q(I) \subseteq f(q')(I)$ we have:

$$pre(q', I) = \frac{f(q')(I) \cap q(I)}{f(q')(I)} = \frac{q(I)}{f(q')(I)}$$

With a similar calculation we have $pre(q'_v, I) = \frac{q(I)}{f(q'_v)(I)}$. We can denote $f(q')$ and $f(q'_v)$ as:

$$\begin{aligned} f(q')(X) : -R_1(X, Y, Z), Y \in \mathcal{PV}(b', adom(B_1)), \\ Z \in \mathcal{PV}(c'_1, adom(C_1) \\ f(q'_v)(X) : -V_1(X, Y, Z, W), Y \in \mathcal{PV}(b', adom(B)), \\ Z \in \mathcal{PV}(c'_1, adom(C)), W \in \mathcal{PV}(d'_1, adom(D)) \end{aligned}$$

By adding more tuples to $R_i, (i \neq 1)$ we build an instance I such that:

$$\mathcal{PV}(b'_1, adom(B_1)) \subset \mathcal{PV}(b'_1, adom(B))$$
$$\mathcal{PV}(c'_1, adom(C_1)) \subset \mathcal{PV}(c'_1, adom(C))$$

According to the retrieval model in Section 3.4, query interface uses the predicted values set $\mathcal{PV}$ to retrieve the tuples. A larger $\mathcal{PV}$ will result in more tuples. Thus, we have $f(q') \subsetneq f(q'_v)$ which leads to $pre(q', I) < pre(q'_v, I)$. □

The view definition V in Theorem 3.5.0.1 contains constants in its head. Thus, the exact rewriting of a query using V, shown by q_v will introduce extra constants to the query. Users may alter q_v by dropping the extra constant to achieve a higher search recall. This technique is called query relaxing. The following corollary shows that the views in Theorem 3.5.0.1 is not recall preserving even after relaxing the query.

Corollary 3.5.0.1. *The view definitions in Theorem 3.5.0.1 does not preserve the recall for relaxed queries.*

Disjunctive views with irreversible rule: The following theorem presents another condition in which using views will decrease precision or recall.

Theorem 3.5.0.2. *Given schema $\mathcal{R}$ and a set of views $\mathcal{V}$, $\mathcal{V}$ is not precision preserving if there is a disjunctive view $V \in \mathcal{V}$ with empty active domain and no constants in view definition head.*

Proof. A disjunctive view definition without constants will have the following general form:

$$V(X) : -R_1(X_1)$$
$$V(X) : -R_2(X_2)$$

where $X \subseteq X_1 \cap X_2$ doesn't contain any constants. Assume $x \in X_1$ and query $q(X) : -R_1(X_1), x = c$. Although this query does not have an exact rewriting using the views, a possible rewriting of this query is $q_v(X) : -V(X), x = c$ where c binds attribute $R_1.A$ and $V.A$. With a similar proof as 3.5.0.1 it can be shown that the larger domain of V will result in $pre(q'_{\mathcal{V}}) < pre(q')$. □

Query Optimization and Effectiveness Trade-off: Given a query, traditional optimizer systems push down the selection operators in q to enhance the efficiency of join operation. This approach can effect the effectiveness of answering q in a noisy channel environment. Applying joins can reduce the $\mathcal{PV}$ for an attribute which leads to a better efficiency.

3.6 Experiments

3.6.1 Experiment Setting

Dataset: We use Freebase database that was created from June 6th, 2013 dump of Freebase[2]. A *topic* in Freebase represents a single concept or real-world entity. A *type* denotes an IS-A relationship about a Topic. For example, Shakespeare is a topic with type Person type i.e. Shakespeare IS-A person. A *property* of a topic defines a HAS-A relationship between the topic and the value of the property. As an example, Paris topic has a population (property) of 2153600 (value)[3]. We extract Person, Film and TV Program types and their properties. We pick the entities that appear at least one time in the query workload and convert each type with a graph structure to a relational database with star schema and store it in a MySQL database. The statistics of our database is shown in Table 3.1.

Query Workload and Execution: We use a sample of MSN query log that is annotated with Freebase topics. In this query log, a semi-automated statistical method is used for annotating the keywords of the query with an attribute (property) from Freebase. These queries contain between 2 to 6 keywords where each query has a related desired answer in our database. For each experiment, we select a set of vague queries as well as non-vague queries. Vague queries are the ones with vocabulary gap between the query and the database. Some queries from our query log and their related answers are shown in Table 3.2. The examples in this table show the vocabulary gap between queries and their answers.

Each query is composed of keywords and their related attributes. We build a SQL query based on the keywords and their annotations in a query [40]. Since we have

[2]https://developers.google.com/freebase/data
[3]http://wiki.freebase.com

Table 3.1: Statistics of relations in the Freebase dataset

relation	number of tuples	size (MB)
person	2751593	1290.27
film	209816	96.55
tv_program	52027	59.56

assumed the user is aware of the schema, we drop the attributes that are related to the schema before submitting it to the database. Consider keyword query "Movie Terminator 1995" over the following schema:

film(id, film_name, film_description, release_year)

tv_program(id, program_name, program_description, release_year)

The attributes of this query are shown in Table 3.3. The *Type* attribute is an information about the schema. We drop that attribute and build the following SQL query:

```
SELECT * FROM film WHERE film_name = 'Terminator' AND release_year = 1995
```

After building this query, we submit it to the database using a retrieval system that is explained next.

Retrieval System: We use Galago[4] for creating inverted indexes for each table. In retrieval process, we use LM ranker as the underlying retrieval algorithm. Given a query, we determine the desired table(s) for answering the query and rank its candidate answers using the ranker algorithm for each table. For queries over union of two or more tables, we use a pre-built materialized view over those tables. There are some other methods to merge the results of different tables in distributed databases such as CORI [23]. However, we do not use these methods to avoid any external source of error in the experiments.

Ranking Quality Metrics: We evaluate the query results using the following metrics: 1) precision-at-3 (P@3) that is fraction of relevant answers among top three retrieved results and 2) reciprocal rank that is computed as the inverse of rank of first relevant answer int the results list. We report the mean reciprocal rank (MRR) of queries in our query workload. In addition, to measure the significance of the results, we report the results of paired t-test at significance level of 5%.

Hardware Setting We run the experiments on a Linux server with 8 Intel(R) Xeon(R)

[4]http://lemurproject.org/galago.php

Table 3.2: Sample MSN queries and their answers

No	Query Text	Answer Topic
1	Good Times TV Programs	Good Times
2	batman	Batman Begins
3	snakes	Snakes on a Plane
4	the death of joseph stalin	Stalin
5	Good Times set come movie	Good Times

Table 3.3: A sample MSN query and its annotations

Keyword	Attribute
Movie	Type
Terminator	Film_Name
1995	Release_Year

3.40GHz cores and 16GB of memory. We have implemented the experiment codes using Java 1.6.

3.6.2 Horizontal Decomposition Experiment

In this section, we describe the design of each experiment, and then we analyze the results.

The goal of this experiment is to show that the size of the active domain of an attribute has inverse relation to the precision of answering queries over a database instance. In this experiment the queries are submitted to instances of different sizes I_1, I_2, I_3, I_4, I_5. The first instance just contains the relevant answers to queries. The other instance are selected in a way that $I_1 \subset I_2 \subset I_3 \subset I_4 \subset I_5$. The schema used in this experiment has a single relation that contains information on Films and TV programs:

media(id, name, description, release_year, type).

The precision at 3 (P@3) and mean reciprocal rank of queries are shown in the Table 3.4. As it is shown in the results, the effectiveness of answering queries drops as the size of the database increases. This result supports our assumptions. Consider the following schema $\mathcal{S}$:

$$film(id, name, description, year)$$
$$tv\_program(id, name, description, year)$$

Table 3.4: Impact of active domain size on effectiveness of query answering

Instance	**Size**	**P@3**	**MRR**
I_1	51	0.31	0.94
I_2	2049	0.29	0.83
I_3	20044	0.28	0.68
I_4	30043	0.25	0.66
I_5	261843	0.14	0.44

Table 3.5: Answering queries using views that do not preserve precision

	P@3	MRR
$\mathcal{S}$	0.21	0.56
$\mathcal{V}$	0.14	0.36
p-value	0.003	0.001

We use the *year* attribute and move films and TV programs that are produced after 2005 to a new table called new_release. The films produced before that date have their own table and also the TV programs produced before 2005 have their own table. We define the following views $\mathcal{V}$ over $\mathcal{S}$:

$$film\_old(id, name, description, year)$$
$$: -film(id, name, description, year), year \leq 2005$$
$$new\_relase(id, name, description,' film')$$
$$: -film(id, name, description, year), year > 2005$$
$$tv\_program\_old(id, name, description, year)$$
$$: -tv\_program(id, name, description, year), year \leq 2005$$
$$new\_relase(id, name, description,' tv\_program')$$
$$: -tv\_program(id, name, description, year), year > 2005$$

We submit the queries to the database instance with source schema $\mathcal{S}$ as well as a database instance obtained by view definitions $\mathcal{V}$. The results of answering these queries are shown in Table 3.5. As it is shown in this table, using the view definitions, the P@3 and MRR of query answering drops significantly which confirm the results of Theory 3.5.0.1. The last row of the table shows the p-value for each metric. Both p-values are less than 0.005, which shows the statistical significance of the results.

Next, we evaluate Theorem 3.5.0.2. In this theorem, views definitions increase the ac-

Table 3.6: Answering queries using views without exact rewriting

	P@3	MRR
$\mathcal{S}$	0.21	0.56
$\mathcal{V}$	0.14	0.44
p-value	0.0005	0.0006

tive domain size of the attributes when there is no exact rewriting using views. Consider schema $\mathcal{S}$ from the previous part and view definitions $\mathcal{V}$ as follows:

$$media(id, name, description, year) :-$$
$$film(id, name, description, year)$$
$$media(id, name, description, year) :-$$
$$tv\_program(id, name, description, year)$$

Next we run the queries over $\mathcal{S}$ and $\mathcal{V}$. The effectiveness of answering queries are reported in Table 3.6. The active domain size of attributes of $\mathcal{V}$ are greater than the active domain size of attributes over $\mathcal{S}$ and answering queries over $\mathcal{S}$ results in a higher effectiveness compared to $\mathcal{V}$. This in turn confirms the results of Theorem 3.5.0.2.

3.7 Conclusion

In this chapter, we provided a formal model to investigate the impacts of different schemas on the effectiveness of answering imprecise queries over databases. Our formal framework does not depend on any particular answering algorithm for imprecise queries. We proved that based on the properties of view definitions, imprecise queries may be generally answered more or less effectively using the views.

Chapter 4: Efficient Join Processing using Many-Armed-Bandits

4.1 Introduction

In Section 2.6 and Section 3.6, we presented the results on running time of processing keyword queries and imprecise queries in different database systems. A major bottleneck in processing these queries is the join operator. The join operator is one of the most important, costly and frequent operations over relational databases. It has been, however, a long standing challenge to efficiently join large relations. This challenge is more prominent in interactive systems, where the users expect real-time performance.

The inherent difficulty of processing joins queries is due to the need to inspect all information in the participating relations and find tuples that match the condition of the join. Traditionally, database systems improve the efficiency of join operations by precomputing certain data structures, e.g., indexes, or sorting the relations, e.g., sort-merge join [106]. These methods, however, are not applicable for many use cases where 1) the precompute datastructures are not available and 2) preprocessing the data is costly. For instance, indexes may not be available over some relations in the database or it may take a long time to build one. Similarly, it may take a while or require too much main memory to sort large relations. Moreover, these methods fall short of satisfying user's desired response time for large relations.

To overcome the afformentioned challenges, researchers have proposed join algorithms that process subsets of input relations to provide the users with a sufficiently large subset of answers. More specifically, as opposed to reducing the total time of the join, a group of join algorithms aim at quickly outputting an initial subset of the joint tuples and gradually completing them [2, 64]. This approach enables users to receive and inspect a subset of the answers in a short time, which is useful in many applications, such as interactive data exploration and analysis [44]. A notable example of this approach is *Stop*[25, 26] or *Limit* [123] operators in the database systems, which limit the output tuples of a join operation in a hope to reduce the query response time [2]. These arguments take k as an input and generate k join results in a fraction of time needed to process the whole

join query. The time required to generate k results is defined as the response time and the goal of these systems is to minimize the response time of the queries.

While the state-of-the-art approaches are succesful in lowering the response time, they either require 1) large amounts of memory[59] or 2) require a preprocessed data strucutre or statistics of the underlying data[29]. To overcome these challenges, we propose *bandit join* that efficiently generates k join results by adapting to the underlying data distribution. We model processing a join operator as an online learning problem where the reward is the number of produced joined tuples and the join operator is trying to maximize the produced join results. Unless otherwise noted, in this chapter we focus on equi-join, i.e., joins with equality predicates and leave other types of join as an interesting future direction.

4.2 Related Work

4.2.1 Adaptive Query Processing

Recently, there has been a significant growth in volume and variety of the data stored in database management systems and the users' information needs has become more complex. Data statistics are becoming less available and some of these databases are stored remotely. With these changes, the traditional query processing paradigm of optimize-then-execute becomes insufficient[42]. Adaptive Query Processing (AQP) techniques, address these challenges by adapting their behavior to the characteristic of the data. These methods detect and correct optimizer errors that arises from cost metric simplification and incorrect statistics. In [12] and [42], authors give a comparison of AQP techniques based on different criteria such as plan quality and scalability. They identify the common problem of AQPs and explain the appropriate setting for each technique.

Eddies, a group of adaptive query processing operators, create a data flow of the tuples, route them to the appropriate operator and intercept them when necessary [41, 11]. Eddies get a feecback on the tuple processing rate of each operator and adjust the data flow accordingly. Eddies can choose different operator orderings per tuple basis during the query processing to adapt to the current state. Other works such as [113, 129] have extended this operator to distributed settings. [116] extends Eddies using reinforcement learning techniques. The authors transform the query optimization

problem to an unsupervised learning problem with quantitative rewards. We will cover machien learning in database management systems in more details in this section. Our work is orthogonal to Eddies as our operators learn to change and optimize their intrinsic data access behavior while Eddies focus on inter-operator data flow.

4.2.2 Online Join Algorithms

The goal of traditional join algorithms is to minimize the completion time of a query. These methods are considered blocking methods because their join operator will not generate any results until it processes the whole query. However, in interactive decision support systems, it is important to have non-blocking techniques to achieve real-time performance. There has been a large body of work on non-blocking join techniques. Symmetric hash-join [124] is one of the early non-blocking join techniques. [107] and [118] extend the symmetric hash-join to support n-ary joins. Another extension of these methods is *ripple join*. Ripple join uses a specific read order of the relations to support approximate aggregate queries [59]. This method supports both equijoins and non-equijoins [85]. However, it needs to keep the read tuples in the main memory which makes it an expensive join method.

Some traditional join algorithms answer top-k join queries by sorting the relation which is an expensive operation. It produces a total ordering on the relation whereas the user only needs the top-k tuples. Furthermore, Sorting is a blocking operator and can not be used in pipeline settings. To address the above issues, in [71] the authors propose rank-join algorithm with the above desired properties and implement it based on the ripple join. Rank Join [71] samples tuples from relations and produces the join of the seen tuples so far. It also computes the score of joins using a score combination function. These joins are stored in a queue. At each step, rank join computes a score threshold based on the seen tuples and reports the joins that have a higher score than this threshold. The sampling strategy can impact the performance of rank join. In the paper, authors provide a heuristic to sample the tuples with higher score and show how it improves the performance. The authors implement the rank join based on a hash ripple join (they can also use a block ripple join too). Hash rank join uses a hash table for seen tuples of each relation and a priority queue to store the generated joins and their scores. After scanning a new tuple, hash rank join updates the threshold and inserts the new

tuple into the relevant hash table and uses the other hash tables to produce the feasible joins. Then, it stores the joins and their score in the queue and outputs the joins (from queue) that their score is higher than the threshold. They explain how the join order can impact the performance of hash join and present a heuristic method to pick a good join order and generalize rank join to exploit random access capabilities. Their experimental evaluation in shows that rank join can beat the state of the art J* algorithm.

The idea of *online aggregation* is an example in which the join algorithm gradually produces and updates its output. The subsequent aggregation operator consumes the output of the join operator and computes more accurate estimations as it receives more tuples from the join [59, 85]. Following the same rationale, some join algorithms produce a fixed random sample of the join by processing subsets of the input relations [29]. While these approaches have had great success in improving the join processing time, it is difficult to utilize them in all settings. First, they usually need indexes or pre-computed statistics about the input relations, which are not usually available or take a significant amount of time to compute. For instance, most join algorithms for online aggregation and random sampling need indexes over input relations [85, 102]. Moreover, all current random sampling algorithm over joins need to know several statistics about the input relations [29]. If these statistics or data structures are not available, these algorithms may take a long time to generate a sufficiently large (initial) sample of the output [85].

The goal of authors in [2] is answering aggregation queries very efficiently by building samples of the tables in an offline step and selecting the right sample while answering the queries. There is a spectrum of different approximation techniques based on their efficiency and assumptions. There are approaches that are designed for very specific groups of queries and guarantee consistent and efficient running time (such as sampling methods) and there are methods that do not impose any assumptions on the query workload and suffer from inconsistent running times (such as OLA methods). Our proposed method is an attempt to develop a flexible and consistently efficient method. Their assumption is that queries use the same columns over time. They define these columns as query column sets (QCS). The presented system has two components: 1) Sample building 2) Sample selection. The first component uses stratified sampling technique to build proper sets of samples and the second set uses these sets to answer queries. The submitted queries can either specify an error tolerance rate or a time limit. Based on these inputs, the system decides number of needed samples and uses it to answer the

query.

4.2.3 Parallel Join Techniques

In [39], authors describe the problem of join processing over data streams where we can only store a window of w tuples from each stream. In their paper, joins with a window size of w are considered as ground truth although these joins may fail to produce all the output joining results. However, the problem arises because of the constraint over a buffer memory M which is smaller than the window size w. There are 2 settings for this problem: 1) We have a fast CPU that can join an arriving tuple with everything in the other relations window immediately. In this case, the loss of join is only because of the buffer size M. An algorithm should decide which tuples to keep in the buffer with the objective of minimizing the approximation error; 2) The CPU is slow, thus we may lose results before processing all the tuples in the join buffer. In this case, we need to remove some tuples from the buffer before they get processed. This is called load shedding. In this case, the loss is due to both buffer size and slow CPU which is a more general form. The authors introduce the MAX-subset measure to evaluate the loss of an approximate join. Then, they propose solutions for the following cases: 1) Static Join where the goal is to compute the equijoin of two non-streaming relations. However, k tuples need to be dropped from each relation. They formalize this problem as a bipartite graph and present a DP algorithm to solve it; 2) Fast CPU–Offline: This is the standard window joining method with the strong assumption that we know the tuples that are arriving in the future. For this case, the problem is modeled as minimum data flow over graphs and a solution is provided; 3) Fast CPU–Online: For this case, the authors propose two simple heuristics to evict tuples from the buffers. To do this, they assume for each tuple, we know the probability that it will join with the other relation and use this probability to evict tuples. They also mention finding the probabilities is a challenging problem that has been addressed in online caching.

In [48] authors propose a distributed online join operator. The incoming data from two relations R and S is represented as a matrix where rows are tuples of R and columns are tuples of S and a cell is a corresponding join tuple. They partition the rows into m and columns into n groups, such that $J = m \times n$ where J is the number of physical machines (workers) in their system. Tuple r from R with row group index i is duplicated

and sent to all the worker groups with row index i and any column index. Each worker has a reshuffle task and a join task. During the reshuffle, the received tuple by the worker is sent to the right worker which joins the received tuples if they satisfy the joining condition. The proposed method has an adaptivity loop with three stages: 1) collecting stats; 2) analyzing them; 3) migrating to a new schema if needed. Only one of the workers need to run the adaptivity loop. The goal of migration is to balance the load between workers, so they can process the joins in memory and minimize the chance of using the disk. Migration changes the partitioning schema (m,n) to minimize the load on each worker. The right time for re-partitioning is decided based on some criteria discussed in their paper. Furthermore, the migration transfers the data between workers so it should be done in a way to minimize the transfer overheads. This method is different from ours in that it utilizes distributed data processing and replicates the data to process them in parallel.

4.2.4 Machine Learning in Database Management Systems

Recently, there has been a myriad of approaches to utilize machine learning techniques in designing different components of database managements systems. These approaches take advantage of data characteristics and specific applications to provide highly specialized data management systems [80]. Previous approaches utilize distribution of data to solve different problems such as database tuning [112] and index selection [117, 30]. However, the more recent approaches use machine learning to design different components of the database management systems [80].

In [81], Kraska et al. propose learned database indexes that can adapt to the underlying data distribution and outperform a traditional B-Tree index in specific cases. In their follow-up work, Kraska et al. propose SageDB [81], a database system that its core parts are learned components. The main idea of the paper is to learn some models of the distribution of the data and workload and then learn algorithms and data structures for the learned distributions. This idea is beyond the current use of machine learning for configuration customization, selecting algorithms and self-design. The authors briefly explain how the data distribution models can be learned and then present its use cases in: 1) Data access: single and multi-dimensional indexing; 2) Query execution: sorting, joins and scheduling; 3) Query optimization: learning join orders. Although the main

use case of the proposed methods is for read-heavy OLAP systems, the authors briefly discuss how to extend their idea to update-heavy cases as well.

In [75], authors address the problem of operator selection for query optimization. Given a high level query operator such as convolution or join, there are many physical operators (as hash-join or sort-merge join) to execute the logical operator and based on parameters, such as size of the data, one of the physical operators can outperform the rest. The authors model the above problem as a multi-armed bandit problem where each operator is an arm and the reward is the running time. Then, they use the Thomson sampling method to solve the bandit problem.

Marcus and Papaemmanouil propose a method that models join order enumeration as an RL problem [92]. In their method, states are matrices of relations, their joins, and join attributes. Actions are defined as picking two relations (or two join results) to join together. The reward is the inverse of the final cost of a generated join plan. They train a neural network using PPO method [110] that takes a state as the input and predicts the score of each action. They train the model on episodes generated from 100 queries and test it on a separate set of queries.

In [115], the authors present a learning system called SkinnerDB to find the optimal join order for a query without having previous training data. The paper uses UCT, a tree based Monte Carlo method [79], to learn the optimal join order by bounding its cumulative regret. In their method, the authors form a state tree where each state is a table and a path from root to leaf represents a join order. A score is assigned to each node which depends on the number of times the node has been visited and the average reward of the paths that include this node. Note that only the leaf nodes produce rewards. The generated reward of a leaf node is used to estimate the value of each internal node that is on the path from the leaf to the root node. There are two motivations to use the UCT approach in join order optimization problem: 1) the state space is large; 2) using cumulative regret, one can combine planning and learning phases into a single phase. The proposed naive method is to pick a small batch of an initial table and use UCT to select a path in the join tree. Then, the join is executed in a limited time. If the join terminates successfully, value of every node on the path from root to current leaf is increased by one. If the join does not terminate in a specified timeout threshold, the path is assigned zero reward. The authors enhance this naive method by a traditional query planner. Also, they propose a third method that is the fastest and is based on

the idea to use a different query execution engine that is tailored for their RL needs. SkinnerDB is different from Marus and Papaemmanouil's work in that it learns and plans simultaneously. More specifically, SkinnerDB does not have a separate training phase. Rather, for each given query, it learns as it produces the join output. In this sense, SkinnerDB does not need to generalize anything to build a model. It acts more like a smart tree search method.

Ortiz et al. propose a method to learn states for query optimization [103]. Their objective is: 1) to learn a useful representation for each sub-query; 2) to use the learned representation to find an optimal query plan using reinforcement learning. For the first part, they train a neural network. The input of this network is a selection query and database features such as minimum and maximum values of the attributes. The output of the trained neural network is the predicted values of the cardinality of the input query. Then, they use this neural network to represent any query-database pair as a vector. In the second part of their work, they introduce the idea of using reinforcement learning to learn a query plan and discuss the pros and cons of rewarding after each action and rewarding in the end of an episode.

None of these methods consider the selection operator and our proposed method is an orthogonal approach to the mentioned techniques.

4.2.5 Many-Armed Bandit Algorithms

Many armed bandits are a class of bandit problems where the number of arm pulls n is much less than the number of the arms k. In these problems, beside exploration-exploitation of the seen arms, one should also decide when to discover new arms. In [20], authors assume a Bernoulli distribution on the expected rewards. Based on this assumption, three algorithms are proposed: 1) one-failure where the current arm is pulled until the first failure; 2) m-run strategy uses one-failure until either m continuous rewards are received or m arms are visited. Then, it uses the best arm; 3) m-learning uses the one-failure method during the first m runs and then picks the best arm. These algorithms require the knowledge of the horizon (number of the arm pulls) to compute the optimal value of m.

In [122], authors propose an anytime method that does not require the knowledge of the horizon. They consider a stochastic distribution on the mean reward of the arms

and present epsilon optimality results that depend on the mean reward distribution. The authors first present the UCB-V algorithm that is a variation of the original UCB algorithm. Then, they apply this approach to many armed bandit problem in two settings. In the first setting, K, the number of arms to pull is given. They sample K arms and apply UCB-V. Assuming they know the horizon, the optimal value of K can be found and the results guarantee a $log(n)\sqrt{n}$ regret bound. In the second setting, the authors do not have the knowledge of the horizon. Thus, at time t, they try a new arm based on some established conditions.

Bandit algorithms have been applied to other problems such as top-k retrieval. In [84], authors present an online learning method to identify the top-k most attractive items in retrieval problems. Their method is based on Cascade Model (CM). Cascade Model is a user behavior model which assumes the user examines the ranked list from top to bottom. As soon as the user finds an attractive item, it stops examining the rest of the list. In this setting, the probability of items in position i and j being attractive is independent of each other. Cascading Bandits is an online learning method based on bandits that finds the top-k attractive items at time t. Each item can be considered an arm and the reward is derived from the user feedback. In this setting, the rewards are considered stationary. At each round, the algorithm computes the UCB value of each item and picks the top-k items based on their UCB value. After receiving users feedback, the algorithm updates the attraction probability of each item based on the previous average attraction and the current reward. This probability and some other parameters are used to compute the UCB values. Non-stationary Cascading Bandits addresses the same problem as above in a setting where rewards are non-stationary. They authors propose two approaches for this problem. The first one uses a discount factor on the old attraction probability to tackle the challenge of non-stationary rewards. The second approach uses a sliding window to address the same problem.

4.3 The Online Learning Framework

In this section, we will outline the online learning framework for processing join operators. For the sake of simplicity, we describe the framework for binary join of two base relations. The binary join operator is preceded by two scan operators over two relations. Each scan operator reads blocks of tuples from its corresponding relation on the disk. After reading

a block, the scan operator outputs tuples of this block which is consumed by the join operator as an input. The join operator collects tuples from two relations and checks to see if the received tuples satisfy the join condition. If so, their corresponding joint tuple will be generated as an output, otherwise the tuples will be discarded. Ideally, the scan operators should send tuples that have a higher chance of joining with each other which depends on the values of their join attributes. The scan operators should also avoid sending a tuple that all its joint tuples have already been produced by the join operator. This way, the number of I/O and disk accesses that are the dominant overhead of performing a join will be minimized.

If the tuples of the two relations are arranged on the disk in certain orders, e.g., sorted in the order of the join attributes, scan operators may coordinate their reads easily. Nevertheless, the underlying relations are not usually arranged in such a desired order. We propose an online learning method where the scan operator learns an effective strategy in emitting tuples that have a higher chance of generating a joint tuple. In the remainder of this section, we will cover the details of the proposed learning framework.

4.3.1 Agents, Actions, and Rewards

Agents: Each scan operator is an agent in our learning framework. At each iteration of processing the join, each scan operator reads a tuple from its base table and sends it to the join operator. We use the word iteration and *round* interchangabily to refer to a single iteration of the join. For a binary join of two relations, we assume one of the scan agents is learning to emit tuples that produce a join with higher likelihood which in turn leads to higher efficiency in producing k join results. We assume the second scan operator emits random tuples and does not learn any strategies to emit tuples. If a pair of the input tuples from scan operators satisfy the join predicate and generate a new joint tuple, the join operator outputs their corresponding joint tuple.

We note that the database systems usually read tuples of the base tables in blocks. However, to simplify our framework, we assume that each scan operator reads the information of its underlying table tuple by tuple. We will discuss the extension of our framework where information is read in blocks in Section 4.5. The task of the join operator in our framework is minimal and mainly consists of checking the join condition for the tuples sent from the scan agents and producing and outputting the joint tuples.

Assume that we like to join relations R and S denoted as $R \bowtie S$. R is defined as the outer relation and S is defined as the inner relation. By the abuse of notation, unless otherwise noted, we refer to the scan operators over R and S simply as agent or operator R and S, respectively. In our framework, agent R is the learning agent and agent S acts randomly and sends random tuples. We denote the number of tuples in relation R as $|R|$. For the purpose of brevity, we refer to the tuples of relation R as *R-tuples*.

Actions: We assume that the scan operator is able to perform both sequential access and random access to the tuples on disk. More specifically, each scan operator may perform one of the following *actions* at each iteration of processing the join:

- `next` which reads the next tuple of the input relation on the disk, i.e., sequential read.
- `go` that reads the tuple of the input relation at a given address. The read tuples are sent to the join operator.
- `end` that sends an empty tuple to the join operator when the scan operator reaches the end of the relation in the sequential scan.
- `reuse` in which the agent informs the join operator to use the last sent tuples.

Note that the latter two actions are introduced to simplify the exposition of our framework. An operator may perform other types of actions if auxiliary data structures, e.g., indexes, have been already created on its underlying relation. We, however, assume that such data structures are not available to the scan operators. The operators may perform their actions in parallel or in a fixed order, e.g., one operator sends its tuple to the join operator before the other one.

Reward: An action performed by the learning scan operator is successful if it leads to generating a new joint tuple. The join operator will inform the learning operator of the *success* or *failure* of its action. We set the reward of an action in each round to 1 if the action leads to a new output tuple in that round and 0 otherwise. Other measures of reward are also possible, e.g. assigning values between 0 and 1 for approximate joins. But, our framework uses the simple 0-1 result.

The *reward* of an agent in round T of the learning process is $u_T = \frac{1}{T}\sum_{t=1}^{T} r(t)$, where $r(t)$ is the reward of the action of the agent in round t. For a fixed T, the larger the reward is, the more joint tuples the agent generate.

The number of iterations, T, depends on the underlying application of the join. It may be explicitly given as the input when the goal is to produce the maximum number of joint tuples within a given time [2]. On the other hand, it might be defined implicitly, where the users would like to stop the join after generating a certain number of tuples [25, 26, 64]. In this case, both agents know in which round to stop performing the join based on the success feedback from the join operator. The higher value for the reward, u, leads to faster join response time. If the user wants to view the full join results, the agents should figure out in what round the join operator has produced all joint tuples to stop the join. Subsequent rounds of the join only reduces the rewards of the agents as they do not produce any new results. We will discuss the methods for the agents to identify the number of rounds in this case in Section 4.4. Similar to the above cases, in this case greater value for the reward function will result in a faster response time for the join.

Another interesting reward function is the one that puts relatively greater value on the joint tuples produced early in the join process to encourage faster delivery of the join result [56]. One may use exponential discounting in the reward definition $u = \sum_{t=1}^{T} \delta^t r(t)$, and $0 < \delta \leq 1$. Smaller values of δ encourage generating more output tuples early in the join. On the other hand, the larger values of δ aim at generating more total tuples in T rounds of the join. For example, $\delta = 1$ represents the objective of the traditional join algorithms.

4.3.2 Strategies and Adaptation

The *history* of an agent at round t is a sequence of pairs (a_i, r_i), $0 \leq i < t$ where a_i and r_i are the action and reward of the agent at round i of the join, respectively. The *strategy* of each agent at round t is a mapping from its history to the set of actions at time t.

An agent may follow a fixed strategy to perform the join. For example, modeling the nested loop join algorithm from an online learning perspective, the scan operator for the inner relations follows a fixed strategy of performing a *next* action in each round of the algorithm except for the round where it exhaust all tuples in the relation. In this case, it performs an *end* followed by a *go* to the beginning of the relation in the subsequent round. Similarly, the scan operator for the outer relation performs a *reuse* action in all

rounds of the join excepts for the ones where the other agent plays *end* in which the outer agent performs *next*.

Nevertheless, if the underlying relations contain sufficiently many tuples, i.e., the join has a sufficiently large number of rounds, an agent may achieve a higher reward by modifying its strategy during the join. It may leverage its experience from the previous rounds of the join to modify its strategy and get a potentially greater reward for the next round(s). For example, using the history of the join, an agent may learn that tuple t_1 joins with significantly more tuples in the other relation than tuple t_2. Thus, if it sends t_1 to the join operator more often than t_2, it may generate answers faster than the case where t_1 is sent more frequently. Similarly, if t_1 is picked by the agent earlier than t_2, the join operator may be able to produce and output more tuples earlier in the joining process.

Since the success rate of tuples are *not* known at the start of the join, the agent has to learn them while performing the join. Such a learning method may first explore various actions or sequences of actions and then exploit this knowledge to choose promising actions in the later rounds of the join. The key element in this approach is to balance exploration and exploitation. If an agent mostly explores possible sequence of the actions, it may not deliver many joint tuples or it may take a long time to generate the full answer. On the other hand, if the algorithm mostly exploits the knowledge gained from the previous rounds of the join and performs a limited amount of exploration, it may not find the most rewarding sequence of actions and be ineffective in the long run, which in turn leads to an inefficient join.

Each scan may apply some selection conditions on its relation, e.g., filtering all tuples with certain values in some of their attributes, or project out a subset of non-join attributes of its input tuples. The selections and projections will be applied to all tuples of the underlying relations. Since the existence of selection and projection operators do not impact our analysis and results of our approach, we assume that scan operators do not perform any selection or projection on their underlying relations to simplify the framework.

4.4 Learning To Join

4.4.1 Effective & Achievable Strategies

Given a fixed T, an ideal strategy is the one in which the learning operator sends a tuple that has equal join attribute to the tuple sent by the other operator in each round such that their resulting joint tuple has not been produced in the previous rounds and stop the join where such a pair of tuples do not exist. This strategy is very hard to find as the learning operator does not know which tuples other operator is going to send. Agent R and S may deliver a performance close to the one of this strategy by reordering the tuples of R and S on disk based on the values of their join attributes or building index(es) over these relations. These approaches, however, are out of the reach for our operators for the reasons explained in Section 4.1.

Since methods that require preprocessing, such as reordering, sorting, or building indexes, are not available to our operators, the next best option is to adapt the selection of actions to the order sent tuples from the random scan operator. In the absence of any information about the values of join attribute of the tuples on disk, the access method of operators is sequential-scan, i.e., heap-scan. Heap-scan usually guarantees retrieving data in a random order based on the values of the join attribute [64]. Thus, the learning operator may assume that the other operator is sending tuples in a random order. In these situations, the optimal strategy for an operator is to choose always the action that has the greatest reward between all its available actions [127, 10]. Thus, as operator S sends tuples according to the sequential scan of its relation, operator R should choose the tuple that joins with the most tuples in S.

Using this approach, however, the learning operators faces three challenges. First, it does not know the optimal tuple with the greatest reward before processing the full join, therefore, it has to learn it while processing the join. Clearly, it should deliver a reasonably accurate estimate within relatively small number of rounds. Otherwise, this strategy may take as much time as the strategies that examine the join of every possible pairs of tuples, e.g., nested loop. Second, users would often like to receive some results fast, e.g., in interactive data exploration [70, 64, 59, 2]. Since the learning phase may take a while, it may take a considerable amount of time for the operators to generate some results. Thus, the operators should combine learning a good strategy and producing

output tuples in order to satisfy users' need. Third, as soon as all the joint tuples that can be produced using the optimal tuple are generated, this tuple will not deliver any more rewards. Thus, operators have to detect when the possible rewards generated from a tuple has been exhausted. Also, when this happens, they should choose the next best tuple from the remaining tuples and continue the join and receive rewards.

In this section, we propose a learning algorithm that overcomes the mentioned challenges and prove that it is asymptotically effective. The proposed algorithm has two stages. First operator R learns and estimates the tuple $r_{max} \in R$ that has the maximum reward in a relatively small number of rounds. While learning r_{max}, the operator leverages the information it has to produce joint tuples. Then, operator R will send only the learned tuple to the join operator until there is no hope of producing any joint tuples using r_{max}. Next, R removes tuple r_{max} from its set of available actions and repeats the previous steps to learn the next tuple in R with the largest reward and produces its joint tuples. The operator R will continue this process until all joint tuples are produced or it reaches a given maximum number of rounds T.

4.4.2 Learning the Optimal Action

4.4.2.1 Strategies of the Operators

Operator R aims at finding tuple $r_{max} \in R$ that joins with the largest number of tuples in S among all tuples in R. R and S may communicate to estimate such a tuple ideally in a relatively few rounds. Let each tuple $r_k \in R$, $1 \leq k \leq |R|$ have a success probability of p_k. The parameter p_k is *not* known at the beginning of the join to the operators. The tuple r_{max} has the largest success probability p_{max} among all tuples in R with ties broken arbitrarily.

Since operator S reads and sends tuples in a random order, operator R may use the algorithms for the classic *multi-armed bandit* (MAB) problem to the r_{max} [56, 10]. In MAB, an operator has a set of actions, i.e., arms, with unknown probabilities of success. It would like to select one action in each round such that its expected reward is maximized over sufficiently many rounds. The action with the greatest probability of success delivers the maximum expected reward. However, it may take a long time to estimate the probability of success for every action as the operator may have to

try each action many times. This method also reduces the expected reward as many actions may deliver very small rewards. On the other hand, not exploring and trying sufficiently many actions may deliver a sub-optimal estimate of the most rewarding action. Generally speaking, MAB algorithms in each round choose actions based on their observed probability of success up to the current round and the number of rounds passed from their last selections [10]. This way, the algorithm delivers some reward by choosing the actions that are deemed successful according to the current observations and explore other potentially promising actions to improve the cumulative reward in the long-run. There are several MAB problems of which *Upper Confidence Bound* (UCB) algorithms deliver desired asymptotic guarantees and shown to be effective in empirical studies [10, 119, 57].

An adaptation of MAB algorithms for our operators will be as follows. Operator R may use an MAB algorithm, such as UCB, and selects the tuple that has the highest UCB score in each round. Operator S has a fixed strategy doing a sequential scan, which is equivalent to randomly picking and sending a tuple without replacement in each round. The UCB algorithm is guaranteed to learn the most rewarding action in the long run and produces tuples while learning.

Nevertheless, this approach poses a couple of challenges for the operator R. First, MAB algorithms, such as UCB, usually explore every action several times to learn a reasonably accurate estimate of its reward. If relation R has many tuples, the learning may take a long time. As a matter of fact, we have observed that it takes operator R more than ten hours to explore its tuples using UCB for a relation R of about a million tuples. Second, to be efficient, operator R should have immediate access to all tuples of R in order to pick the one with the highest score according to the UCB selection criterion. Since there is not any indices over R, operator R has to construct and maintain a mapping from tuples' addresses to their UCB scores in the main memory. This mapping will be used to select and update the scores of the tuples throughout the join. To select and send each tuple, operator R looks up this mapping, finds the tuple with the greatest score, and performs random access to read and send the tuple from the disk to the join operator. Thus, operator R has to perform a random access to pick each tuple. As every tuple will be accessed multiple times, this method may significantly slow down the join. Moreover, if R contains many tuples, the mapping may have a considerable storage overhead.

To overcome these challenges, operator R may leverage MAB algorithms designed for the agents that have infinitely many available actions [19, 121, 22], a.k.a infinitely many-armed-bandits. Roughly speaking, these algorithms assume the set of available actions is too large to be fully explored. Thus, they aim at effectively estimating the most rewarding action(s) using a sufficiently small random sample of the set of available ones. There are different algotihms to solve infinitely many-armed bandit problems. We set operator R to use an algorithm called *m-run* [19]. Similar to the approach explained in the preceding paragraphs, operator S performs a sequential scan and sends a tuple of S to the join operator in each round. Operator R does a sequential scan and sends each read tuple to the join operator. Since R performs a heap-scan, one may assume that it randomly selects from its available tuples. Operator R maintains a mapping from the scanned tuples' addresses to their total observed rewards in the main memory. As far as the current tuple of R produces a joint tuple, operator R keeps sending this tuple to the join operator without reading any new tuple and moving ahead in the relation R. As soon as the current tuple of R faces its first failure in a round, i.e., *not* producing a joint tuple in the round, operator R moves ahead in its sequential scan and sends the next tuple on the disk to the join operator in the subsequent round.

If there is a tuple that has m consecutive successes, operator R stops its scan and declares that tuple as the estimated one with the maximum reward. Otherwise, it stops scanning and sending new tuples to the join operator as soon as it reads m distinct tuples. In this case, it checks the total rewards of every tuple read from R so far using the mapping maintained in the main memory and picks the one with the greatest reward as the desired tuple where ties are broken arbitrarily. At this point, the *learning phase* of the strategy finishes.

The amount of the storage needed to keep the mapping of the address and reward of each tuple is modest for a sufficiently small m. We denote the estimated tuple with maximum reward as $\hat{r}_{max}$. In the subsequent rounds of the join, operator R will keep sending $\hat{r}_{max}$ to the join operator and operator S will continue the sequential scan of its remaining tuples and sending them to the join operator. The operators will maintain these strategies until operator S exhausts all tuples in relation S. We call the sequences of rounds starting from learning $\hat{r}_{max}$ until operator S is done with all its available tuples the *super round of* $\hat{r}_{max}$.

4.4.2.2 Asymptotic Analysis

An important question regarding the strategy of the operators in the super-round of $\hat{r}_{max}$ is whether it generates a sufficiently large total reward. Equivalently, we would like to know whether these strategies produce sufficiently small rate of failure in which operators spend I/O access costs but do *not* produce any results. To simplify our analysis, we assume that $|R| < |S|$. Because operators plan to follow the same strategy profile to find other highly rewarding tuples of R and compute their joins in the subsequent rounds of the join, the number of rounds of each super round in the join should be in the order of $|S|$. Otherwise, the overall running time of the full join may become equally or less efficient than the nested loop. In other words, we would like operator S to access each tuple of S only once. The following proposition establishes a lower bound on the expected failure proportion of the strategy profiles in which operator S accesses each of its available tuples only once in a heap-scan. We make the simplifying assumption that the probabilities of success for tuples in R denoted as p_ks are independent. We also assume that p_ks are drawn from a uniform distribution in the interval $[a, b]$ $0 \leq a \leq b \leq 1$.

Proposition 4.4.2.1. *Assume an strategy profile in which operator S accesses each tuple exactly once in a random order. The expected failure rate of this super round has a lower bound of* $(1 - b) + (b - a)\sqrt{\frac{2}{|S|}}$.

Proof. The proof directly follows from Theorem 7 in [19]. □

Proposition 4.4.2.1 holds for every strategy for operator R including m-run. To get a clear understanding of the result of the Proposition 4.4.2.1, let $b = 1$ and $a = 0$. Proposition 4.4.2.1 indicates that a lower bound on the expected failure proportion of every strategy is $\sqrt{\frac{2}{|S|}}$. This lower bound comes from the inherent difficulty of learning the most rewarding tuple online while processing the join and the restrictions on the access paths of our operators to their actions, i.e., tuples.

Next, we ask whether the strategy our operators use will achieve an expected failure proportion close to the lower bound of Proposition 4.4.2.1. Since we would like the learning to scale for relation R with numerous tuples, it is reasonable to set m to a sufficiently small value otherwise the learning of $\hat{r}_{max}$ may take many rounds and require significantly many I/O accesses. On the other hand, small values for m may not deliver sufficiently precise estimate of the desired tuple. Interestingly, the following proposition

shows that it is enough to learn from a random sample of R with a modest size in order to learn a tuple that achieves an expected failure proportion close to the lower bound in Proposition 4.4.2.1. As in Proposition 4.4.2.1, we assume that operator S performs a heap-scan of its underlying relation and reads each tuple exactly once.

Proposition 4.4.2.2. *If operator R uses m-run algorithm with $m = \sqrt{|S|(b-a)}$, the expected failure proportion of a super round of the join is less than or equal to $(1-b)+ 2\sqrt{\frac{(b-a)}{|S|}}$ asymptotically.*

Proof. The proof directly follows from Theorem 8 in [19]. □

Again, to get a better understanding of the result of Proposition 4.4.2.2, let $b = 1$ and $a = 0$. In this case, the expected failure is at most $\frac{2}{\sqrt{|S|}}$, ignoring additive and multiplier constants, for $m = \sqrt{|\ S\ |}$. Since binary join is symmetric, we can assign operator R to the larger relation of the two. In this case, operator R will read a significantly fewer tuples than $|R|$ in each super round. If both relations have almost equal number of tuples, operator R will still read a small sample of all the tuples in R to learn the desired tuple r_{max}. If b is considerably less than 1, operator R has to read even fewer tuples than the case where b is close to 1.

4.4.2.3 Practical Considerations

After finding $\hat{r}_{max}$, the operators will produce the join of $\hat{r}_{max}$ and S, denoted as $\hat{r}_{max} \bowtie S$, except for the ones whose S tuples are read by operator S before operator R determines $\hat{r}_{max}$. If users would like to receive the complete join of the underlying relations, each super round must produce the full join of $\hat{r}_{max} \bowtie S$. Thus, after operator S reaches the end of relation S, it starts scanning and sending the tuples from the beginning of S up to the point where $\hat{r}_{max}$ is determined so that the join operator produces the rest of $\hat{r}_{max} \bowtie S$. These additional scans are *not* needed if the user does not need the full join answer and for example requires a relatively small sample of the results [25, 26]. The rounds that start immediately after the learning phase of each super round until all join tuples of the desired R-tuple in the super round are produced constitute the *join phase* of the super round.

The discussed strategies may produce some joint tuples while learning $\hat{r}_{max}$. These tuples can be used to build a part of $\hat{r}_{max} \bowtie S$ which in turn reduces the overall response

time of the join and users may immediately see some results. However, including these joint tuples from the learning phase in the final join results may cause duplicate joins. There are two scenarios that may lead to duplicate joint tuples.

1. During the join phase of the current super round, the joint tuples will be reproduced during the additional scan of the operator S explained in the preceding paragraph.

2. In the next super rounds of the join, the joint tuples whose R-tuples is *not* $\hat{r}_{max}$ but have been generated in the learning phase of the past super rounds.

To resolve these issues, the join operator may maintain the pairs of addresses for R- and S-tuples of the joint tuples produced during the learning of $\hat{r}_{max}$ and uses it to exclude duplicate joint tuples from the final result set. The join operator will maintain and update this table throughout the whole join process. This table has a relatively small space overhead as it keeps only the information of the joint tuples produced during learning of $\hat{r}_{max}$. Because only a relatively small number of tuples are accessed during the learning phase in m-run algorithm, these joint tuples are usually a small fraction of all the tuples generated in each super round and the join process. Moreover, $\hat{r}_{max}$ will be removed from the set of tuples that are available to operator R at the end of its super round. Thus, the information of its produced joint tuples will be removed from the table after its super round finishes. Similar elimination will be applied at the end of each subsequent super round.

To use m-run algorithm, operator R should know the values of b and a before processing the join. These values sometimes are precomputed and stored to be used for query optimization [53]. Nevertheless, these values may not be known to the operators. In our empirical studies reported in Section 4.5, we have set the value of b close to 1 and a to 0. Our empirical studies in Section 4.5 show that using such a generous setting, the proposed strategies still outperform the alternative join methods. As explained in Section 4.4.2.2, to reduce the number of rounds and consequently the time of each super round, we choose operator R to access the tuples in the larger relation. However, in primary to foreign key joins, this approach will not be used if the relation with the foreign key is the largest one as each tuple of that relation joins with exactly one tuple of the other relation. Because learning the tuple(s) with larger reward is not useful for this relation, we set operator R to be over the relation that has the primary key.

The scan operators read tuples of an underlying relation in larger data units of blocks [53]. In our framework, each operator may read from an underlying relation in blocks but passes the information to the join operator tuple-by-tuple following the iterator model of communication in database systems [53]. Nevertheless, one may choose the unit of action to be a group of tuples or blocks. Each group will be sent from operators R and S to the join operator similar to sending single tuples explained in this section. The join operator perform an in-memory join, e.g., in-memory nested loop join, to join the tuple groups. Then, the join operator outputs the joint tuples and reports the reward to the operators. The reward of each group is the number of joint tuples normalized to be between $[0, 1]$. The size of each group depends on the available main memory. Our empirical studies in Section 4.5 indicate that using blocks or groups of tuples as the unit of action during learning may lead to a higher success rate than those of a single tuple. This is mainly because joining groups of tuples or blocks from R and S may have a higher chances of delivering some joint tuples, i.e., positive reward, than that of a single pair of R- and S-tuples. Because the rewards are not binary in this setting, the bounds of Propositions 4.4.2.1 and 4.4.2.2 may not hold. There are strategies with some theoretical guarantees for this setting [121, 22]. For practical reasons, we do not use them for our operators which we discuss in Section 4.4.4.

4.4.3 Strategies for the Full Join

After finishing the super round of $\hat{r}_{max}$, operators may resume the join to find the next most rewarding R-tuple by excluding the learned tuple of the last super round from the actions available to R. Instead of re-running the strategies in the preceding super rounds, the operators leverage the information gained from the previous super round(s) to learn the next most rewarding tuple with a relatively small number of I/O accesses. More precisely, operator R eliminates tuple $\hat{r}_{max}$ from its list of available tuples to expoit, by maintaining a list of addresses of every $\hat{r}_{max}$ in the preceding super rounds. It also removes the entry of this tuple from the mapping that maintains the total reward of each R-tuple. Operator R maintains two pieces of information of the preceding super round. It keeps the number of R-tuples explored in the learning phase of m-run algorithm in the last super round. We denote this value as m_{pre}. It also maintains the address of the last tuple accessed during the learning phase of the last super round, denoted by l_{pre}.

It needs this value to continue the learning phase of the last super round from where it is left off. To perform the learning stage of the m-run algorithm in the current super round, operator R initializes the value m in the strategy to m_{pre} and will scan the tuple immediately after l_{pre} in the first round of the new super round. Operator S maintains the position of the last tuple it has read during the learning phase of the last super round. At the first round of the new super round, it scans the tuple immediately after that position. Operators will follow the same steps as in the last super round for the remaining rounds of the current one.

Since the total rewards of R-tuples during the learning phase of m-run algorithm may be considerably less than m, the number of tuples read by the operator in the learning phase of the current super round may be significantly fewer than m. Thus, the learning phase in the super round following the first one may have a relatively small time overhead. If the goal is to efficiently produce a given number of joint tuples, the algorithm will stop as soon as the desired number of tuples are produced [25, 26]. If the user would like to find as many joint tuples as possible in a certain number of rounds, i.e., time limit, the algorithm will stop as soon as it reaches the given number of rounds. Otherwise, it continues until all joint tuples are produced. Towards the end of the join, operator R may not have enough available tuples to follow the m-run strategy as it may have found the significantly rewarding tuples and generated their joins. As soon as operator R detects this situation, the operators may simply follow a fixed deterministic strategy without learning, such as nested loop, for the remaining rounds of the algorithm.

If there is enough main memory available, operators may reduce the rounds of the join and subsequently the running time of the join by learning a list of rewarding tuples instead of only one in a super round. In this method, operator R and S will follow the learning phase of k consecutive super rounds without performing their subsequent join phases. At the end of this learning stage, operators will learn the k most rewarding tuples in R. Then, the operators perform a single join phase for all the learned k tuples by joining them with tuples in S the same way it is explained in this section. This methods will reduce the number of scans of S almost by a factor of k.

Due to the inherent randomness of sampling and learning, the guarantees provided by Proposition 4.4.2.2 are correlated with the expected failure and equivalently success proportions of the discussed strategies. One may find cases where a deterministic and fixed strategy, such as nested loop, outperforms the aforementioned strategies. An ob-

vious example is the one where the tuples in R are sorted based on their total reward. Choosing R as the outer relation, the (tuple-based) nested loop method will need fewer I/O accesses than our approach to generate the output. Our goal in this paper is to devise methods that deliver a superior performance in average case which may not be always the most efficient method.

4.4.4 Strategies for Non-Binary Rewards

Yizao Wang et al. have proposed MAB algorithms [121] for infinitely many-arms with rewards between 0 and 1. They provide assymptotic bounds on the expected failure rate of their method similar to what we have discussed in Section 4.4.2. However, their algorithm requires an operator to keep the order of $\sqrt{\mid S \mid}$ in the main memory, which is not usually possible. Thomas Bonald et al. have proposed an algorithm for MAB with infinitely many-arms and proved that it achieves the lower bound of $\sqrt{2n}$ for sufficiently many trials of a successful arm [22]. Although this algorithm guarantees higher asymptotic reward for a single super-round, it has a couple of issues to be used in our setting. First, it may need significantly more than $\sqrt{n}$,i.e., $m\sqrt{\frac{n}{2}}$ trials with large values for m to find successful arms. Thus, using this technique, the operators may need significantly more I/O accesses to find and commit on an optimal tuple in each super-round of the join. Second, as opposed to our chosen method, operator R may access different number of tuples in each super-round. Therefore, it is not clear how to efficiently reuse the results of the explorations in the preceding super-rounds for the future rounds.

4.5 Experiments

We evaluate our method against nested loop and block nested loop join [53]. We do not compare against sort-merge and hash join as they contradict our assumptions of 1) small memory availibility and 2) no pre-processing of data.

Table 4.1: Tuple Counts of Tables

Scale	Table		
	part	lineitem	Order
$s = 1$	200,000	6,000,000	1,500,000
$s = 2$	400,000	12,000,000	3,000,000
$s = 3$	600,000	18,000,000	4,500,00

4.5.1 Experiment Setting

4.5.1.1 Dataset and Queries

We use TPC-H [1] to generate the queries and databases for our experiments. TPC-H is a benchmark for decision support systems and is widely used to evaluate query processing and optimization techniuqes. It contains a set of business oriented queries and their database and illustrates the systems that process large volumes of data to find the answer of critical business oriented questions. Figure 4.1 shows the TPC-H schema. The database generator of TPC-H takes different parameters such as scale/size of the data and distribution of the attributes. We experiment with 3 differnt database scales. Table 4.1 shows the number of tuples of different database scales that we use throughout the experiments. Note that the first columns shows the scale parameter feeded into TPC-H database generator. We use TPC-H queries with a binary join in them.These queries are specified as Q12 and Q14 in TPC-H benchmark. Note that, we only process and evaluate the join part of these queries and drop their aggregate function as processing the extra operators may introduce noise to the measure running time. We evalute each query over different databases sizes and with different result size k.

```
Q12: SELECT * FROM order JOIN lineitem ON o_orderkey = l_orderkey;
Q14: SELECT * FROM part JOIN lineitem ON p_partkey = l_partkey;
```

Besides TPC-H, we evaluate our algorithm on real world Wikipedia database. The details of this dataset is provided in Section 2.3.2. We define a query over the `article` and `article-link` tables as:

[1] available at: `http://www.tpc.org/tpch/`

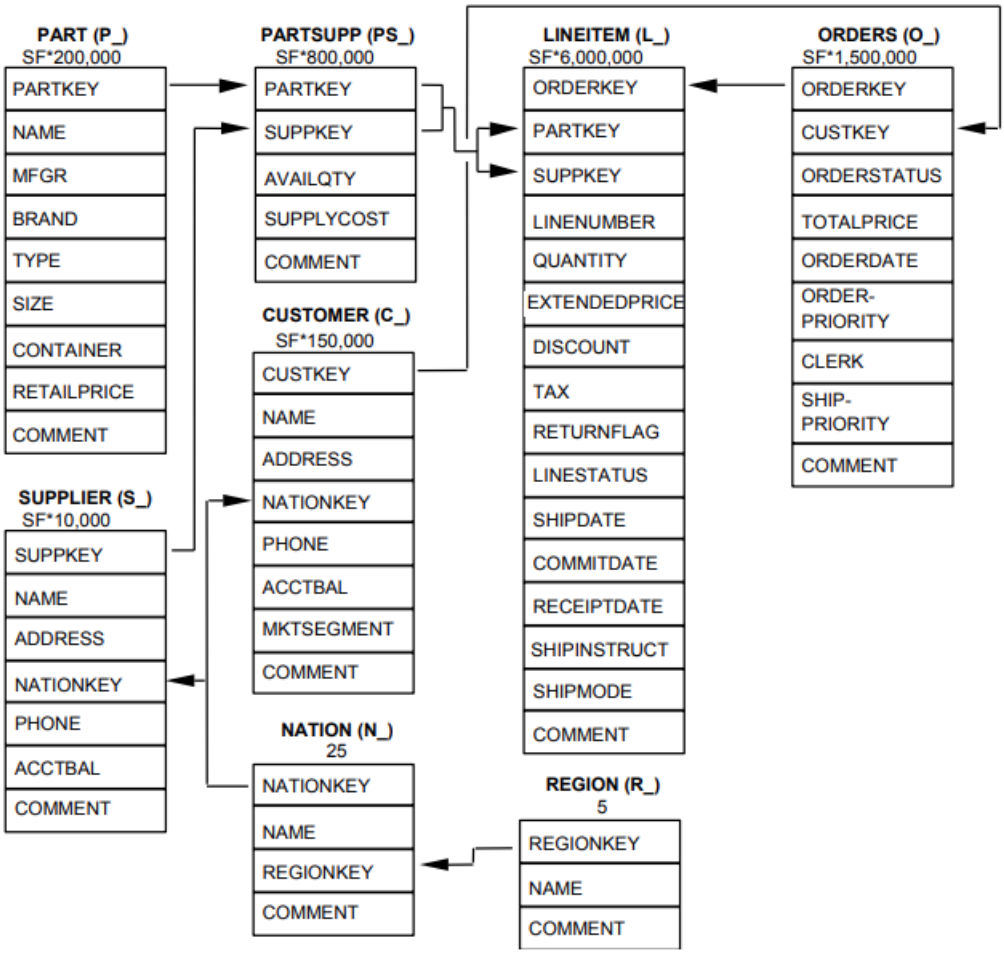


Figure 4.1: TPC-H Schema from [37].

```
QW: SELECT * FROM article JOIN article_link
        ON article.id = article_link.article_id;
```

4.5.1.2 Implementation, Hardware Setup, and Operating System

We implement bandit join inside PostgresSQL 11.5 database management system. We compile and run the database servoer on a Linux server with Intel(R) Xeon(R) 2.30GHz cores, 500GB of memory, 100 TB of disk space and CentOS 7 operating system. Multi-threading is turned off for the database server by setting the `max_parallel_worker` parameter to zero. PostgresSQL has two main parameters to control its memory consumption. First one, `work_mem`, is the amount of memory that is used per operator. For example, if an operator needs to build an in memory data structure, it can not exceed `work_mem`. Second one, `shared_buffers`, is the size of the cache that is used by PostgresSQL. We set both of `work_mem` and `shared_buffers` to their minimum possible values, 64KB and 128KB respectively.

Bandit join algorithm needs to have both sequential and random access to tuples of R. More specifically, when bandit join is in exploration phase, it reads tuples sequentially. However, in exploit phase, the algorithm needs to do one random access to the tuple (or tuple group) with the highest reward. To implement a method that has both random and sequential access, we define a btree index on a dummy generated attribute of R. Using this index, we can have both sequential and random access to tuples of R without providing the join operator benefits of having an index on join attribute.

One can use the `LIMIT` operator to set the value of k in the experiments. However, this operator leads to a query plan with extra nodes beside the join operator which impacts the run-time. Thus, to measure the accurate run-time of the join, we define a cursor over the result-set of each query and iterate over the result set and measure the time to obtain k results. This is implemented using Python 3.6.4 and psycopg2 library.

Unless otherwise noted, we set the group size of bandit join to 32. This number guarantees that bandit join will not keep more than one disk block of R in the main memory.

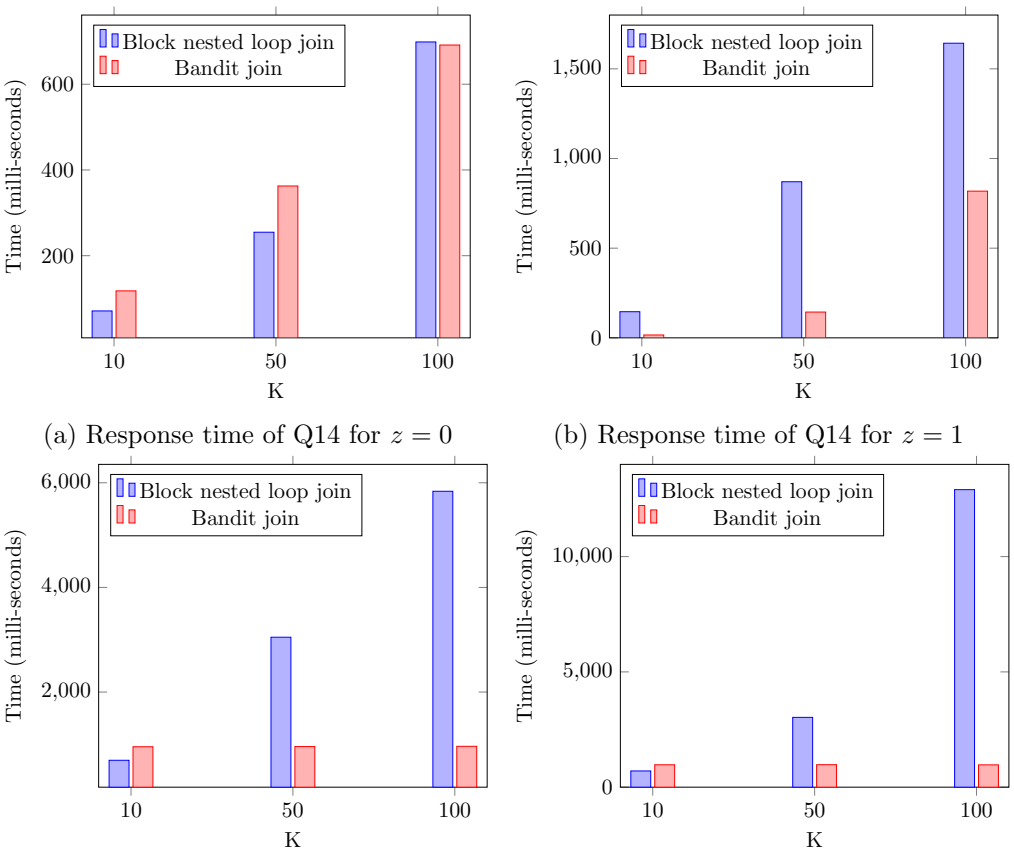
4.5.2 Evaluation of Bandit Join against Block Nested Loop Join

In this section, we compare the performance of bandit join to block nested loop join. We implemented block nested loop join as an improved version of PostgresSQL's nested loop. In block nested loop, instead of reading one tuple from R and joining it with S, we read a group of tuples from R and join it with S. In other words, we change the scanning paradigm from single tuples to groups of tuples. This reduces the IO access of the nested loop join if tuples of R are not clustered on the disk.

One of the parameters that impact the query processing time is the probability distribution of the attribute values and more specifically the skew in data [48, 108, 100, 125, 86, 83]. In this section, besides datasets with different sizes and values of k, we evaluate the query run-time over datasets with different skews. More specifically, we assume a zipfian distribution with parameter z over the data [48]. Setting $z = 0$ will result in uniform distribution. As we increase the value of z, the distribution becomes more skewed.

The skew in the value of join attributes will impact the join selectivity. If the skew is equal to zero, (ex. when we have a uniform distribution), the join selectivity of different tuples will be similar to each other and the range of join selectivities will be small. However, if the skew is high, the join selectivity of different tuples will have a high variance and a large range. In this case, the m-run algorithm will be able to efficiently identify an optimal tuple or tuple group. This in turn will improve the asymptotic bounds that we have discussed in Section 4.4.2.2.

Figure 4.2a shows the running time of Q14 using block nested loop and bandit join over a dataset with zero skew (uniform distribution). As shown in the figure, for $z = 0$, block nested loop outperforms bandit join for $k = 10$ and $k = 50$. The reason is that, with uniform distribution, it is difficult for bandit join to learn which tuple group can produce more joint results. However, as we increase k, bandit join has more time to learn and outperforms block nested loop. Figure 4.2b shows the same results for datasets with $z = 1$. This figure shows that for a slightly skewed data, the bandit join is doing much better than block nested loop. The reason is, for this distribution, the bandit join is able to learn a proper tuple group. On the other side, block nest loop has a high chance of getting unlucky and facing "useless" tuple groups, until it finds a group that actually generates sufficiently enough join results.



(a) Response time of Q14 for $z = 0$

(b) Response time of Q14 for $z = 1$

(c) Response time of Q12 for $z = 0$

(d) Response time of Q12 $z = 1$

Figure 4.2: Response time of bandit join compared to block nested loop join for different values of k

Besides the response time of the query to generate k join results, we are also interested in measuring the average time of generating each join tuple in the result set based on their order. More specifically, it is important to generate the first results faster than the later ones. To measure this criteria, we use the discounted average of time to generate each result. Consider a set of join results as $J = \{j_1, \ldots, j_k\}$ where it takes t_i time to generate result j_i. Using a positive discount factor $\gamma < 1$, the discounted average is defined as:

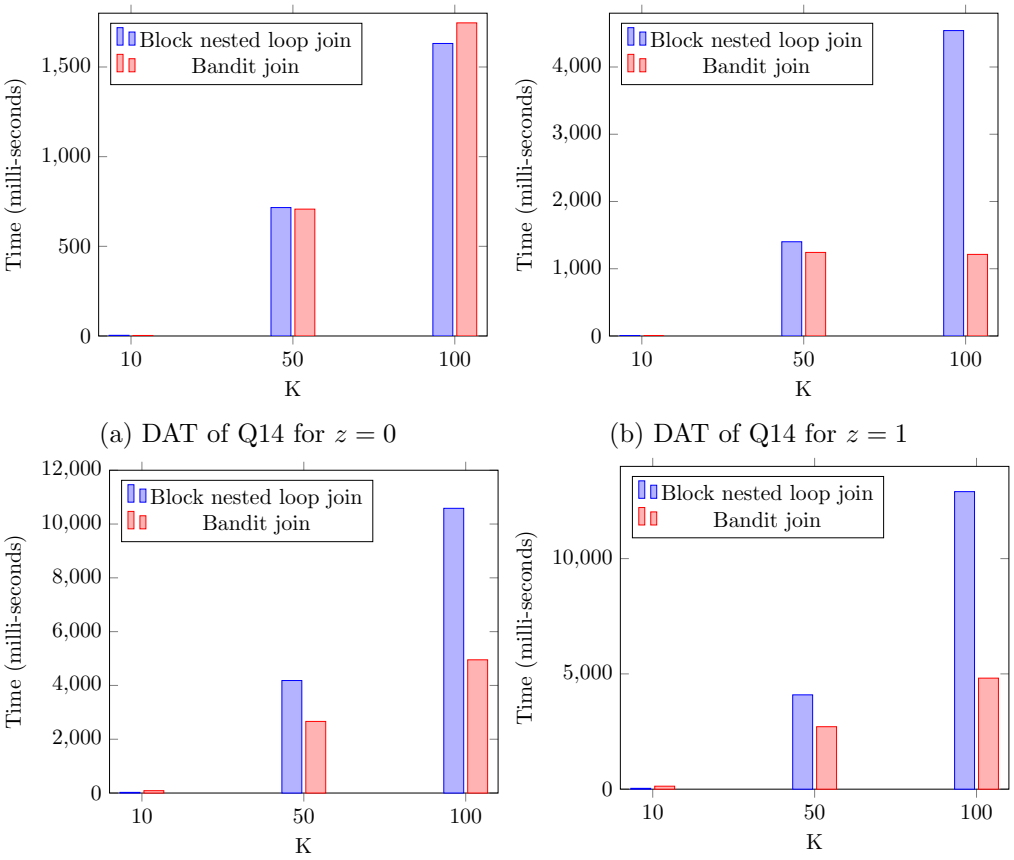
$$\textit{discounted-average(J)} = \sum_{i=1}^{k} \gamma^i t_i$$

This metric, puts a higher weight on the first generated results and the weight decays as we generate more results. As an example, if the discount factor γ is equal to 9, then the time to generate the first result is 0.9^9 times more important than the time that it takes to generate the 10^{th} result. Since decay of 0.9^i is very fast, it mostly reflects the average of first few run-times in the list. To extend it to the whole list, we define a step size of 10 and increase the power of discount after every 10 results (instead of each result). This way, the discounted average will still value the first few results more but would decay with a slower pace compared to the original metric. Figure 4.3a and 4.3b show the discounted average time (DAT) of generating k results for datasets with $z = 0$ and $z = 1$ respectively. The results are consistent with the results of response time.

Figure4.4 shows the response time results of QW over the wikipedia database. The response time of QW is reported for different values of k. As shown in the figure, bandit join outperforms the baseline when $k > 10$.

4.5.3 Scalibility of Bandit Join

Next, we evaluate the impact of database size on the performance of bandit join and block nested loop. The size of these databases are provided in Table 4.1. We run Q14 and Q12 on each of them and report the results. Figure 4.5 show the response time of bandit join and nested loop join on three databases with different sizes. We see that as the database size grows, bandit join outperforms block nested loop with a larger margin. Note that Q12 is a primary key to primary key join. In this setting, m-run algorithm can not learn the optimal tuple/group. However, because it will skip a tuple/group after first failure, it still outperforms block nested loop in generating k results.



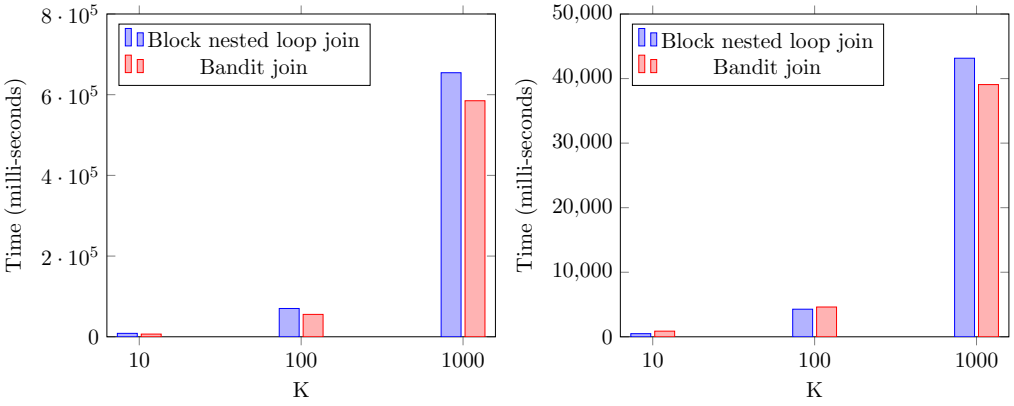
(a) DAT of Q14 for $z = 0$

(b) DAT of Q14 for $z = 1$

(c) DAT of Q12 for $z = 0$

(d) DAT of Q12 for $z = 1$

Figure 4.3: Discounted average time (DAT) of bandit join compared to block nested loop join for different values of k



(a) Response time of QW (group size = 64) (b) Response time of QW (group size = 256)

Figure 4.4: Reponse time of bandit join compared to block nested loop join for *QW*

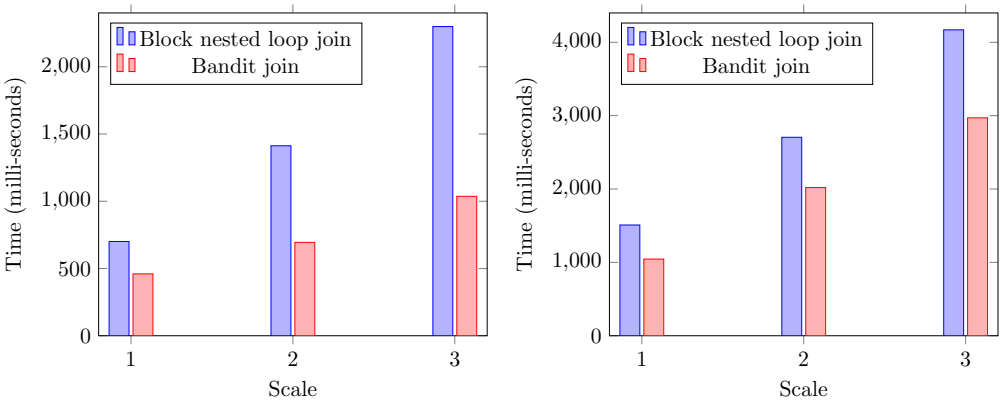
Figure 4.6 shows the discounted average time of processing Q12 and Q14 using bandit and block nested loop joins. These results are consistent with the results of response time discussed in the previous paragraph.

4.5.4 Skew Resilience

Generally, as the skew in data increases, the performance of bandit join becomes much better than block nested loop. However, there is an optimal point for for performance of bandit join based on data skew. More specifically, if the z is too small or too large, may have slighly worse performance than a medium value for z. This characteristic is more obvious in resopnse time of Q14 shown in Figure 4.7a and 4.7b. We can see for Q14 the optimal z is 2. This value can change based on the dataset size and other parametrs.

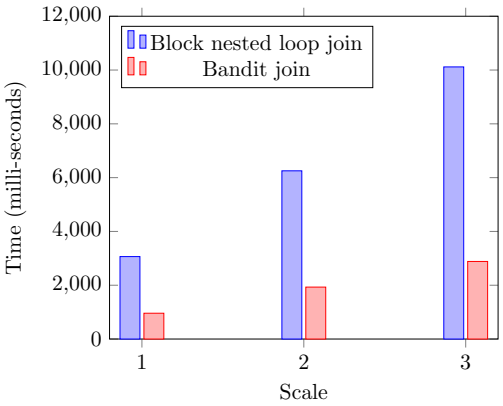
4.5.5 Evaluation of Bandit Join against Nested Loop Join

In this section, we evaluate bandit join against nested loop join. We run Q14 on TPC-H generated database. Figure 4.8a compares the run-time of bandit join to nested loop join using k as the independent variable (over x-axis) for $z = 0$. Figure 4.8b shows the same results for $z = 1$. For all databases sizes, we see that bandit join out performs nested loop for different values of k. Figure 4.9a and 4.9b shows the discounted average time of

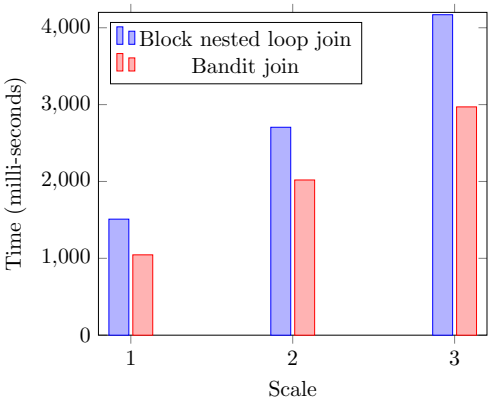


(a) Response time of Q14 ($k = 50$)

(b) Response time of Q14 ($k = 100$)

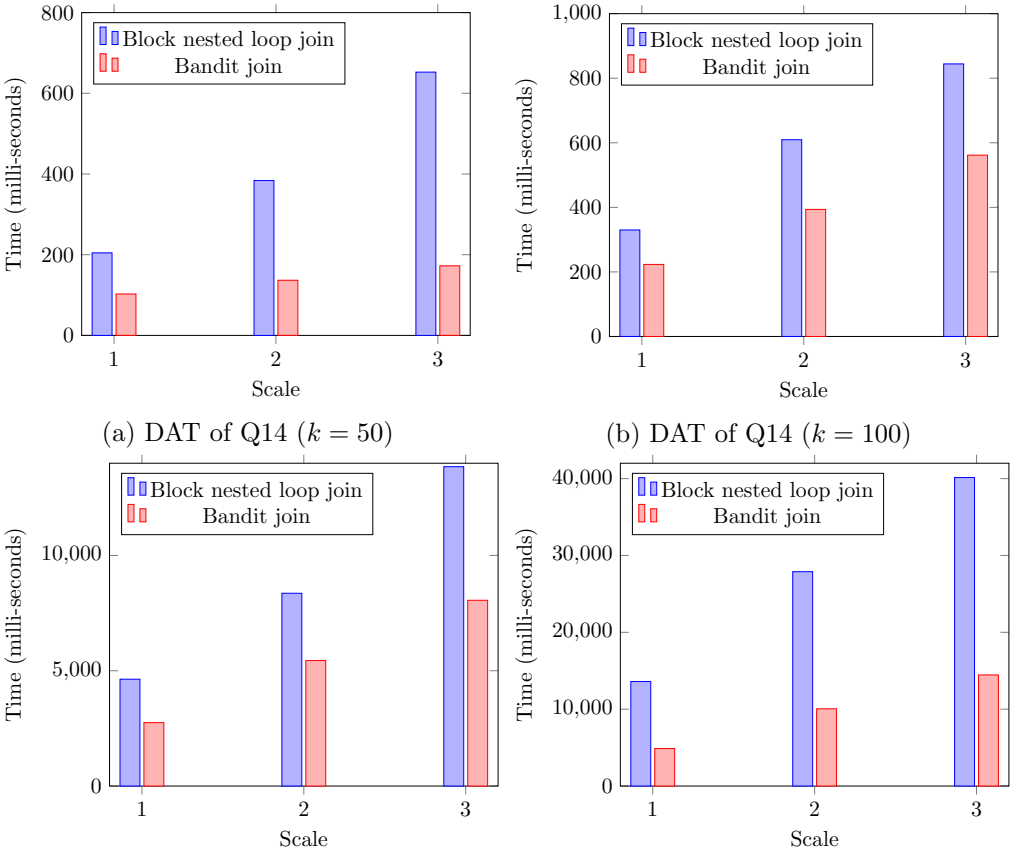


(c) Response time of Q12 ($k = 50$)



(d) Response time of Q12 ($k = 100$)

Figure 4.5: Impact of database size on the response time of bandit join and block nested loop join



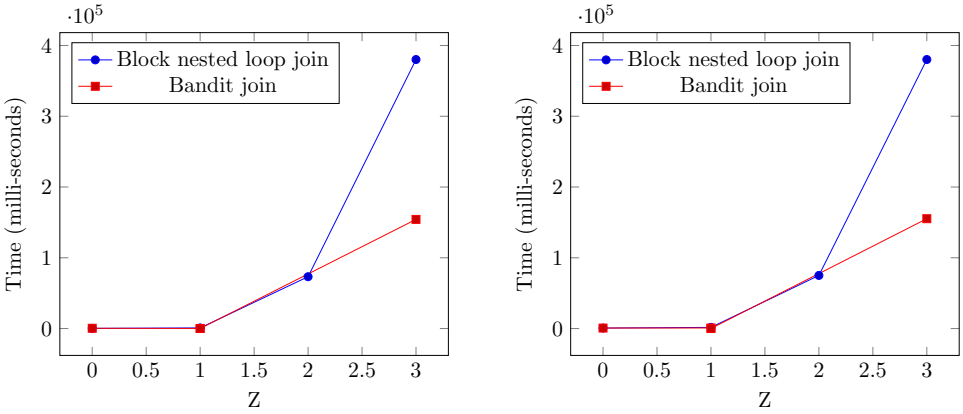
(a) DAT of Q14 ($k = 50$)

(b) DAT of Q14 ($k = 100$)

(c) DAT of Q12 ($k = 50$)

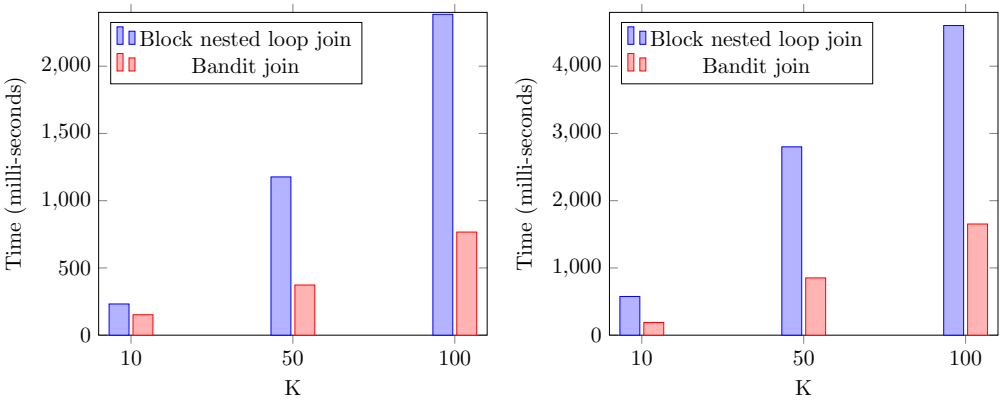
(d) DAT of Q12 ($k = 100$)

Figure 4.6: Impact of database size on discounted average time of bandit join and block nested loop join



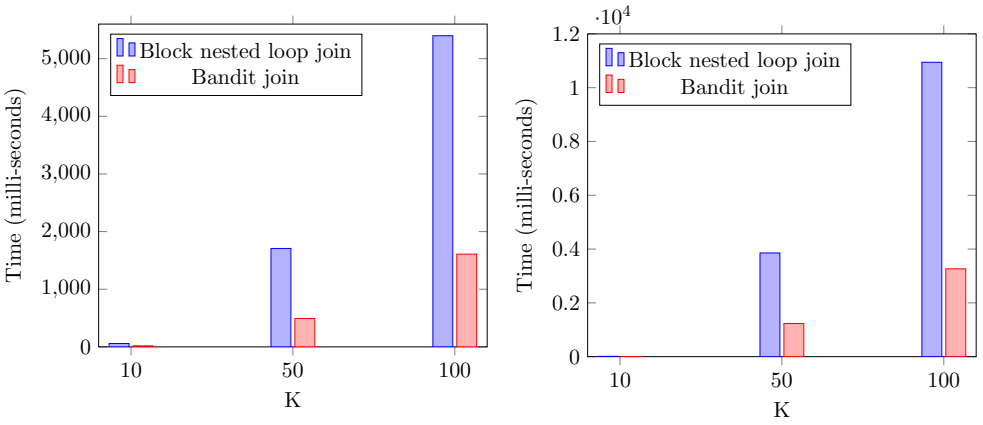
(a) Response time of Q14 ($k = 50$) (b) Response time of Q12

Figure 4.7: Impact of data skew on the response time of bandit join and block nested loop join ($k = 100$)



(a) Response time of Q14 for $z = 0$ (b) Response time of Q14 for $z = 1$

Figure 4.8: Response time of bandit join compared to nested loop join for different values of k



(a) DAT of Q14 for $z = 0$

(b) DAT of Q14 for $z = 1$

Figure 4.9: Discounted average time (DAT) of bandit join compared to nested loop join for different values of k

Q14 using bandit join and nested loop join. We can see bandit join outperforms nested loop join in both cases. The reason for these results is that for one tuple group of R, nested loop scans all of S and then proceeds to the next tuple group. If the first tuple group in R does not produce many join results, then nested loop wastes a lot of time on this tuple group. However, bandit join gives a limited change to each block. If it does not produce any joins, it proceeds to the next block leading to better run-time results.

4.6 Conclusion

In this chapter, we have proposed bandit join, an online learning method for join processing and execution with low reponse time in certain settings. More specifically, we were interested in the domains where the database system cannot afford to build extra data-structures or perform preprocessing prior to the query execution. The objective of the learned operator is to minimize the response time of the join, i.e. the time it takes for the operator to generate a subset of the final results. The response time is one of the main evaluation metrics for realtime performance of many systems such as interactive systems. We have reviewed the current join processing approaches in the mentioned domain and described their similarities to and differences from our proposed method and finally evaluated our technique against a widely used baseline. Our empirical studies

shows the scenarios where the proposed bandit join method outperforms the baseline. Furthermore, the results show the scalibility and skew resilience of the proposed method.

There are many interesting future directions to follow this line of work. In our proposed method, relation R is the learning agent and relation S is a random agent. Extending this framework such that both relations can learn, in a coordinated game setting, can futher improve the efficiencly of the bandit join. With this extension, the operators will have two different criteria in picking the optimal tuples: 1) Which tuple can generate more results; 2) Which tuple can transfer more information to the other operator to help them carry out their learning process. In this sense, the framework will transform into a collaborative learning setting.

Furthermore, it would be nice to extend the work to support top-k queries. In this setting, the input tuples of the join have scores and the join operator is given an aggregate function to compute the score of a joint tuple. The goal of such an operator is to produce joint tuples that have the higher score than the rest. This operator is useful in ranked retrieval settings such as the one introduced in Chapter 2.

Chapter 5: Conclusion and Recommendation for Future Research

5.1 Summary

The objective of this dissertation is to show it is possible to redesign traditional and highly trusted components of a database system with the goal of providing effective and efficient query processing. The structure, size and distribution of the underlying data in a DBMS can be utilized to redesign data access and query processing methods to increase the effectiveness and efficiency of these methods.

In Chapter 2, theoretical and empirical results on impact of database size on the effectiveness of keyword processing are presented. We provided the conditions under which database size negatively impacts the keyword search effectiveness and how one can overcome this issue. Next, we presented a system that uses these results to provide effective and efficient keyword query search over relational databases.

Chapter 3 introduced the idea of modeling imprecise query search as a noisy communication channel. We described how the complexity of a database schema can impact the effectiveness of answering imprecise queries. We presented theoretical results that can be used to asses if one schema may deliver a higher search effectiveness than the other. Finally, we verified these results using empirical studies on real world data and queries.

Finally, in Chapter 4, we described the problem of processing join queries efficiently in the absence of pre-built data structures and pre-processing steps. Then, we presented an online learning method to improve the response time of join query processing compared to the state-of-the-art.

5.2 Recommendations for Future Research

This work can initiate some exciting future research directions that we briefly present here. The provided system in Chapter 2 is designed based on the stationary distribution of access to a database items. However, there are many scenarios that the access

frequencies posses a non-stationary distribution. We propose an extension to current presented system that models and utilizes the non-stationary access frequencies.

In Chapter 4, we presented an online learning method to improve the efficiency of scan and join operators. In the presented framework, each operator tries to learn an optimal strategy without collaborating with other operators. One interesting future direction for this line of work is to investigate the frameworks where the operators learn optimal strategies by collaborating with each other. The collaboration of operators can further increase the efficiency of the query processing over very large databases that can not be queries in real-time using the current techniques.

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley, 1994.

[2] 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.

[3] Joseph Albert, Y. E. Ioannidis, and R. Ramakrishnan. Conjunctive Query Equivalnce of Keyed Relational Schemas. In *PODS*, 1997.

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

[5] 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.

[6] 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.

[7] Timothy G. Armstrong, Alistair Moffat, William Webber, and Justin Zobel. Has Adhoc Retrieval Improved Since 1994? In *SIGIR*, 2009.

[8] Timothy G. Armstrong, Alistair Moffat, William Webber, and Justin Zobel. Improvements That Don't Add Up: Ad-Hoc Retrieval Results Since 1998. In *CIKM*, 2009.

[9] Paolo Atzeni, Giorgio Ausiello, Carlo Batini, and Marina Moscarini. Inclusion and Equivalence Between Relational Database Schemata. *TCS*, 19, 1982.

[10] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[11] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 261–272, 2000.

[12] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research (CIDR), Jan. 2005*, 2005.

[13] 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.

[14] 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.

[15] D. Barbosa, J. Freire, and A. Mendelzon. Designing information-preserving mapping schemes for XML. In *VLDB*, 2005.

[16] C. Beeri, A. Mendelzon, Y. Sagiv, and J. Ullman. Equivalence of Relational Database Schemas. In *STOC*, 1978.

[17] 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.

[18] 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.

[19] Donald A. Berry, Robert W. Chen, Alan Zame, David C. Heath, and Larry A. Shepp. Bandit problems with infinitely many arms. *The Annals of Statistics*, 25(5):2103–2116, 1997.

[20] Donald A Berry, Robert W Chen, Alan Zame, David C Heath, Larry A Shepp, et al. Bandit problems with infinitely many arms. *The Annals of Statistics*, 25(5):2103–2116, 1997.

[21] 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.

[22] Thomas Bonald and Alexandre Proutière. Two-target algorithms for infinite-armed bandits with bernoulli rewards. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, pages 2184–2192, USA, 2013. Curran Associates Inc.

[23] James P Callan, Zhihong Lu, and W Bruce Croft. Searching distributed collections with inference networks. In *SIGIR*, 1995.

[24] 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.

[25] Michael J. Carey and Donald Kossmann. On saying “enough already!” in sql. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 219–230, New York, NY, USA, 1997. ACM.

[26] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 158–169, 1998.

[27] 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.

[28] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic Ranking of Database Query Results. In *VLDB*, 2004.

[29] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On Random Sampling over Joins. In *SIGMOD*, 1999.

[30] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007.

[31] 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.

[32] Yi Chen, Wei Wang, Ziyang Liu, and Xuemin Lin. Keyword Search on Structured and Semi-structured Data. In *SIGMOD*, 2009.

[33] E. F. Codd. Further Normalization of Database Relational Model. *Database Systems, Prentice-Hall*, 1972.

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

[35] 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.

[36] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSearch: A Semantic Search Engine for XML. In *VLDB*, 2003.

[37] Transaction Processing Performance Council. Tpc benchmark$^{\text{TM}}$h decision support standard specification, 2018.

[38] 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.

[39] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD*, 2003.

[40] Elena Demidova, Xuan Zhou, Irina Oelze, and Wolfgang Nejdl. Evaluating Evidences for Keyword Query Disambiguation in Entity Centric Database Search. In *DEXA*, 2010.

[41] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 948–959, 2004.

[42] Amol Deshpande, Zachary G. Ives, and Vijayshankar Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[43] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, 1999.

[44] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: An automatic query steering framework for interactive data exploration. In *SIGMOD*, 2014.

[45] 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.

[46] 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.

[47] Oliver M Duschka and Michael R Genesereth. Answering recursive queries using views. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 109–116. ACM, 1997.

[48] Mohammed Elseidy, Abdallah Elguindy, Aleksandar Vitorovic, and Christoph Koch. Scalable and adaptive online joins. *Proceedings of the VLDB Endowment*, 7(6):441–452, 2014.

[49] Ronald Fagin, Benny Kimelfeld, Yunyao Li, Sriram Raghavan, and Shivakumar Vaithyanathan. Understanding Queries in a Search Database System. In *PODS*, 2010.

[50] Ronald Fagin, Benny Kimelfeld, Yunyao Li, Sriram Raghavan, and Shivakumar Vaithyanathan. Rewrite Rules for Search Database Systems. In *PODS*, 2011.

[51] Ronald Fagin, Phokion G Kolaitis, Renée J Miller, and Lucian Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1), 2005.

[52] Wenfei Fan and Philip Bohannon. Information Preserving XML Schema Embedding. *TODS*, 33(1), 2008.

[53] Hector GarciaMolina, Jeff Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2008.

[54] Vahid Ghadakchi and Arash Termehchy. There is no dichotomy between effectiveness and efficiency in keyword search over databases. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1675–1676. IEEE, 2018.

[55] Vahid Ghadakchi, Arash Termehchy, and Abtin Khodadadi. Answering keyword queries using effective subsets. In *31st International Conference on Scientific and Statistical Database Management*. ACM, 2019.

[56] J. C. Gittins. *Multi-armed Bandit Allocation Indices*. Wiley, Chichester, NY, 1989.

[57] Artem Grotov and Maarten de Rijke. Online learning to rank for information retrieval: Sigir 2016 tutorial. In *Proceedings of the 39th International ACM SIGIR*

Conference on Research and Development in Information Retrieval, SIGIR '16, pages 1215–1218, New York, NY, USA, 2016. ACM.

[58] Ramanathan Guha, Rob McCool, and Eric Miller. Semantic search. In *Proceedings of the 12th international conference on World Wide Web*, pages 700–709. ACM, 2003.

[59] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 287–298, New York, NY, USA, 1999. ACM.

[60] Alon Y Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[61] Mark Hall, Hugo Huurdemann, Mette Skov, David Walsh, et al. Overview of the inex 2014 interactive social book search track. 2014.

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

[63] 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.

[64] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 171–182, New York, NY, USA, 1997. ACM.

[65] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB*, 2003.

[66] 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.

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

[68] R. Hull. Relative Information Capacity of Simple Relational Database Schemata. In *PODS*, 1984.

[69] R. Hull and S. Abiteboul. Restructuring Hierarchical Database Objects. *TCS*, 62(1), 1988.

[70] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of data exploration techniques. In *SIGMOD*, 2015.

[71] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB JournalThe International Journal on Very Large Data Bases*, 13(3):207–221, 2004.

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

[73] Magesh Jayapandian and H. V. Jagadish. Automated Creation of a Forms-based Database Query Interface. In *VLDB*, 2008.

[74] 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.

[75] Tomer Kaftan, Magdalena Balazinska, Alvin Cheung, and Johannes Gehrke. Cuttlefish: A lightweight primitive for adaptive query processing. *arXiv preprint arXiv:1802.09180*, 2018.

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

[77] 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.

[78] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. SnipSuggest: Context-Aware Autocompletion for SQL. In *VLDB*, 2011.

[79] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[80] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. 2019.

[81] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[82] 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.

[83] M Seetha Lakshmi and Philip S. Yu. Effect of skew on join performance in parallel architectures. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*, pages 107–120. IEEE Computer Society Press, 2000.

[84] Chang Li and Maarten de Rijke. Cascading non-stationary bandits: Online learning to rank in the non-stationary cascade model. *arXiv preprint arXiv:1905.12370*, 2019.

[85] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629. ACM, 2016.

[86] Wei Li, Dengfeng Gao, and Richard Thomas Snodgrass. Skew handling techniques in sort-merge join. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 169–180. ACM, 2002.

[87] 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.

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

[89] 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.

[90] 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.

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

[92] Ryan Marcus and Olga Papaemmanouil. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, page 3. ACM, 2018.

[93] Xiangfu Meng, ZM Ma, and Li Yan. Answering approximate queries over autonomous web databases. In *Proceedings of the 18th international conference on World wide web*, pages 1021–1030. ACM, 2009.

[94] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *VLDB*, 1993.

[95] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. Schema Equivalence in Heterogenous Systems: Bridging Theory and Practice. *Information Systems*, 19(1), 1994.

[96] Ion Muslea. Machine Learning for Online Query Relaxation. In *KDD*, 2004.

[97] Ullas Nambiar and Subbarao Kambhampati. Answering imprecise queries over web databases. In *Proceedings of the 31st international conference on Very large data bases*, pages 1350–1353. VLDB Endowment, 2005.

[98] Arnab Nandi and Jagadish H. V. Effective Phrase Prediction. In *VLDB*, 2007.

[99] Arnab Nandi and H. V. Jagadish. Assisted Querying Using Instant-Response Interfaces. In *SIGMOD*, 2007.

[100] Hung Q Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *arXiv preprint arXiv:1310.3314*, 2013.

[101] 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.

[102] Rank Olken and Doron Rotem. Simple Random Sampling from Relational Databases. In *VLDB*, 1986.

[103] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. *arXiv preprint arXiv:1803.08604*, 2018.

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

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

[106] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[107] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 353–364, 2003.

[108] Microsoft Research. Program for tpc-h data generation with skew.

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

[110] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

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

[112] Vamsidhar Thummala and Shivnath Babu. ituned: a tool for configuring and visualizing database parameters. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1231–1234. ACM, 2010.

[113] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 333–344, 2003.

[114] Thanh Tran, Peter Mika, Haofen Wang, and Marko Grobelnik. Semsearch'11: the 4th semantic search workshop. In *WWW*, 2011.

[115] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. Skinnerdb: regret-bounded query evaluation via reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1153–1170. ACM, 2019.

[116] Kostas Tzoumas, Timos Sellis, and Christian S Jensen. A reinforcement learning approach for adaptive query processing. *History*, 2008.

[117] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*, pages 101–110. IEEE, 2000.

[118] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 285–296, 2003.

[119] Aleksandr Vorobev, Damien Lefortier, Gleb Gusev, and Pavel Serdyukov. Gathering additional feedback on search results by multi-armed bandits with respect to production ranking. In *WWW*, pages 1177–1187. International World Wide Web Conferences Steering Committee, 2015.

[120] Qiuyue Wang and Andrew Trotman. Data-Centric Track. In *INEX Workshop*, 2010.

[121] Yizao Wang, Jean-Yves Audibert, and Rémi Munos. Algorithms for infinitely many-armed bandits. In *Proceedings of the 21st International Conference on Neural Information Processing Systems*, NIPS'08, pages 1729–1736, USA, 2008. Curran Associates Inc.

[122] Yizao Wang, Jean-Yves Audibert, and Rémi Munos. Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, pages 1729–1736, 2009.

[123] Michael Widenius, David Axmark, and Kaj Arno. *MySQL reference manual: documentation from the source.* " O'Reilly Media, Inc.", 2002.

[124] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[125] Yu Xu and Pekka Kostamaa. Efficient outer join data skew handling in parallel dbms. *Proceedings of the VLDB Endowment*, 2(2):1390–1396, 2009.

[126] 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.

[127] H. Peyton Young. *Strategic Learning and Its Limits.* Oxford University Press, 2010.

[128] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword Search in Relational Databases: A Survey. *IEEE Data Engineering Bulletin*, 33(1), 2010.

[129] Yongluan Zhou, Beng Chin Ooi, and Kian-Lee Tan. Dynamic load management for distributed continuous query systems. In *Proceedings of the 21st International*

Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan, pages 322–323, 2005.