

© 2020 Mohit Vyas

INDEX DESIGN FOR INFORMATION RETRIEVAL APPLICATIONS USING  
DATABASE CONCEPTS

BY

MOHIT VYAS

THESIS

Submitted in partial fulfillment of the requirements  
for the degree of Master of Science in Computer Science  
in the Graduate College of the  
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Kevin Chang

## ABSTRACT

Integrating database (DB) and information retrieval (IR) technologies has long been an important problem. Recent growth in tagged textual data, which is a combination of structured knowledge bases and unstructured text data, has made principled index design for IR applications even more desirable. In this work, we propose a framework for designing IR applications using DB concepts. We model an IR application as a specialized DB system that is required to support only a fixed predefined query workload rather than general SQL queries. The physical design of an information retrieval system is optimized for a prespecified target query workload. We then identify IR indexes as basic building blocks of the design of an IR application. An IR index is formalized as a look-up function built over a materialized view. The overall "index design problem" is then to find an index design with lowest cost from a space of candidate designs. Since the space of candidate designs can be doubly exponential in the size of query workload, we give polynomial time heuristic algorithm with provable guarantees for a weighted set cover based relaxation of the original problem. This is then combined with an overarching branch-and-bound based optimality preserving state-space search algorithm that efficiently prunes the state-space by using the above heuristic algorithm. Finally, we show via experiments how the proposed framework can be used to obtain index designs for different kinds IR applications in practice.

*To my parents, for their love and support.*

## ACKNOWLEDGMENTS

I would like to thank my supervisor, Professor Kevin Chang, for his invaluable guidance throughout the process.

## TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION . . . . .	1
CHAPTER 2	SYSTEM MODELING . . . . .	4
2.1	Query Workload . . . . .	4
2.2	Index Tables . . . . .	8
2.3	Answering Queries Using Index Tables . . . . .	9
CHAPTER 3	OPTIMIZATION FRAMEWORK . . . . .	14
3.1	Candidate Index Designs and Cost Model . . . . .	14
3.2	Branch-and-Bound Search . . . . .	17
CHAPTER 4	WEIGHTED SET COVER BASED RELAXATION . . . . .	22
4.1	Context Independent Cost . . . . .	22
4.2	Two-Level Weighted Set Cover Problem . . . . .	24
CHAPTER 5	RELATED WORK . . . . .	31
CHAPTER 6	EXPERIMENTS . . . . .	33
CHAPTER 7	CONCLUSION AND FUTURE WORK . . . . .	41
REFERENCES	. . . . .	42

## CHAPTER 1: INTRODUCTION

Inverted index [1] has long been a method of choice for large scale search application. Its simplicity and specialization to the search task allows it to be much more scalable than any DB based implementation [2]. However, there has been considerable growth in both quantity and variety of data in form of RDF knowledge bases [3][4][5][6], and combination of text and RDF data [7][8][9]. Although inverted index works great for traditional keyword based search applications on unstructured textual data, it is an ongoing work to extended its design for the novel information retrieval applications like entity search [10][11][12] and keyword search combined with RDF data [13][14][15]. The design of many of the systems for these novel applications bear resemblance to the inverted index design. For instance, [11] describes two indexes: (i) Document-inverted index, and (ii) Entity-inverted index, for the entity search problem. An entity search [10] query typically requires a list of entities that appear in the same document as the keywords in the query. The document-inverted index described in [11] executes entity search by mapping documents to entities much like how inverted index maps keywords to documents. Entity-inverted index [11] goes one step further and directly maps the input keyword directly to the entities co-occurring with it across the corpus. In the similar vein, systems [15][13] for SPARQL queries [16] combined with text based queries uses an index similar to entity-inverted index of [11] for the text part. For the SPARQL queries, a common approach [17] is to store six different copies of data sorted in the six possible orders of columns subject, predicate and object namely (SPO, SOP, PSO, SOP, OSP, OPS). Again, much like how keyword is mapped to documents in inverted index design, in the RDF index with order, say SPO, we map subjects to predicates followed by objects. The main goal and motivation for this work is to abstract out and formalize the idea of an index design of an IR application, using DB concepts, in order to reason about these index designs within a common well defined framework. The proposed framework for index design, built using database concepts, serves the following important goals:

- *Separate implementation of an IR index from problem specification it is trying to solve:* A common framework allows us to reuse implementations as long as the problem specification remains the same. For instance, the systems described in [18], [15], and [13] all loosely have the same problem statement of combining full-text search with SPARQL queries. But since there is no common way to represent the system specification, their implementations are incompatible with each other.

- *Define a space of possible index designs and pick the one with least cost in a principled manner:* For instance, it allows us to answer question like: Are the document-inverted index and entity-inverted index, proposed in [11], only possible index designs for entity search? What are other possibilities or index designs and which index design is preferable over other in a given circumstance?
- *Combine an IR application with a database system in a seamless manner:* integrating DB and IR technologies is a central problem [19][20][21] as many applications require the capabilities of both general purpose querying on structured data and text based IR queries like traditional keyword search or entity search. Modeling an IR application, using DB concepts, makes it easier to combine it with an RDBMS system.

Inspired from [2], we model an IR application as a specialized database. As argued in [2], an inverted index can be seen as a relation  $KD(text, document)$  with columns *text* and *document* where each record, say ["cat", wikipedia.org/Cat], stores the information of a text value "cat" appearing inside the document wikipedia.org/Cat. However, unlike a database system, the keyword search system only needs to support keyword to document queries and not the other way round. As a result, inverted index can sort the values of the relation  $KD$  w.r.t the column *text*. Furthermore, since insertions and deletions in a search system are infrequent, they can be compressed and augmented with specialized data structures like skip lists [1] offline in order to execute the keyword search query efficiently. Finally, unlike database, a search engine does not have to worry about issues ACID properties and security. These specializations allows for many optimization making search index implementations, like Lucene [22], much faster as compared to one based on RDBMS implementation. We generalize this idea in this work and show how index designs, for various IR applications like entity search and text search combined with RDF [23], data can be seen as specialized database applications.

In chapter-2, we formalize the "index design problem" for IR applications. Chapter-3 lays out the overall optimization framework including the space of possible index designs and a cost model to quantitatively compare different designs for the same specification. We show that the proposed problem is NP-hard and the space of possible index designs exponential. Since the space of possible index designs can be exponential, we propose a branch-and-bound [24] based algorithm in section-3.2 to efficiently prune out sub-optimal candidate solutions using lower and upper bounding heuristics. The key advantage of a branch-and-bound based strategy is that it can utilize efficient heuristics without losing the optimality of the algorithm. Upper and lower bounding heuristics form a key component of a branch-and-



bound based algorithm [24]. To that end, we propose a weighted set cover based relaxation of the original problem which allows for an efficient greedy algorithm [25] with  $O(\log(n))$  factor approximation. In chapter-4, we discuss how the original problem can be relaxed into two weighted set cover formulations, whose optimal solutions provide upper and lower bounds for the original problem. We then obtain a greedy algorithm for this relaxed version with an  $O(\log^2(n))$  factor approximation. Finally, we conduct experiments in chapter-6 to show how the design of various IR applications can be modelled within the proposed framework and compare the different strategies to obtain the least cost index design.

## CHAPTER 2: SYSTEM MODELING

Inspired from [2], we model an Information Retrieval (IR) application as a specialized database that is optimized for a specific read-only query workload. In chapter-6, we show how this formulation can be used to model a variety of IR applications including entity search, RDF graph queries and read-intensive RDBMS applications. Physical design of an IR application is composed of data structures called "index tables" that are optimized for the target query workload. Defined in section-2.2, an index table is a look-up function defined over the contents of a materialized view. In section-2.3, we show how the queries can be rewritten using index tables. Next, we modify the classical notion of a query plan to represent different ways in which a rewriting of a query, in terms of index tables, can be executed. Finally, we define the problem of designing an IR application as selecting which indexes to store and how to use them efficiently to answer the queries in the workload.

### 2.1 QUERY WORKLOAD

In this section, we will formalize the requirements of an IR application as a workload of read-only queries that it needs to support. As a representative example, we will look at the keyword search system that uses inverted index to retrieve documents containing query keywords. As shown in [2], we can model a collection of documents using the following schema:

$$\mathcal{C} = \{K(kId, text, idf), D(url, dId, len), KD(kId, dId, tf)\} \quad (2.1)$$

Here relation  $K$  stores the list of all keywords present in the corpus along with its inverse-document frequency (IDF) " $idf$ " and keyword ID " $kId$ ". Similarly, relation  $D$  stores the list of documents with their URLs and lengths. Relation  $KD$  stores the occurrence information of a keyword in any document. For instance, if a keyword, say, "ferrari" with  $kId = 42$  appears 6 times in a document with  $dId = 35$ , then the record  $[42, 35, 6]$  will be present in the relation  $KD$ . The attributes  $tf$ ,  $idf$  and  $len$  can be used for computing the BM25 [1] score shown below:

$$BM25(Q, D) = \sum_{k \in Q} IDF(k) \frac{TF(k, D)(k_1 + 1)}{TF(k, D) + k_1(1 - b + b \cdot \frac{|D|}{avgdl})} \quad (2.2)$$

Here  $TF(k, D)$  and  $IDF(k)$  are term-frequency and inverted document frequency [1] respectively.

Now say the requirement for this system is to efficiently compute the BM25 scores for all possible keyword queries, like "red ferrari". To compute the scores, we first need to collect the ingredients (TF, IDFs and document lengths) required to compute the score in equation-2.2. Then we need to add the contributions from each keyword to obtain the scores for each document. Finally, we need to rank order the documents based on the scores. The first part of collecting the ingredients can be formulated as a conjunctive query [26] over the corpus  $\mathcal{C}$  as follows:

$$\begin{aligned} Q_k(idf_1, idf_2, url, dId, len, tf_1, tf_2) :- & K(kId_1, "red", idf_1), \\ & KD(kId_1, dId, tf_1), K(kId_2, "ferrari", idf_2), \\ & KD(kId_2, dId, tf_2), D(url, dId, len) \end{aligned} \quad (2.3)$$

A conjunctive query  $Q$  [26], written as  $Q(\vec{H}) : -g_1(\vec{X}_1), g_2(\vec{X}_2), \dots, g_n(\vec{X}_n)$ , is composed of sub-goals, variables and constants. Each sub-goal  $g_i(\vec{X}_i)$  represents an assertion that the list of constants and existential variables  $\vec{X}_i$  is present in the underlying relation  $g_i$ . For instance, the sub-goal  $K(kId_2, "ferrari", idf_2)$  asserts that the record  $[kId_1, "ferrari", idf_2]$  is present in the relation  $K$ . By taking logical "AND" or all sub-goals, we obtain a select-project-join query whose joins are represented by shared variables and selections are represented using constants. For example, the above conjunctive query represents the following select-project-join query:

```
SELECT K1.idf, K2.idf, D.url, D.len, KD1.dId, KD1.tf, KD2.tf
FROM K as K1, KD as KD1, K as K2, KD as KD2, D
WHERE K1.text = "red" AND K2.text = "ferrari" AND K1.kId = KD1.kId
      AND K2.kId = KD2.kId AND KD1.dId = KD2.dId = D.dId
```

For more details on conjunctive queries, we refer the reader to [26]. In this work, we only consider conjunctive queries for which all variables are head variables. For example,  $Q(kId) : -KD(kId, dId)$  is a valid conjunctive query which represents the set of keyword IDs that are associated to at least one  $dId$ . However, we don't include such queries in our framework as the variable  $dId$  is not a head variable. Main reason behind this design choice to keep the exposition simple. Now, the computation of scores and preparing a ranked list can be modelled using "group by" and "order by" operations respectively. In this paper, however, we only focus on the first part of the above query execution that corresponds to retrieval of the "ingredients" required to compute the scores. We exclude score computation

and rank list generation from our framework due to a couple of reasons. Firstly, one might want to support different scoring methods like TF-IDF, vector space models, learning to rank, etc. that use the same ingredients. An optimization specific to a scoring function would limit the scope of the system to that particular scoring function. Secondly, much of the research in the area of materialized view selection has been for conjunctive queries. This however prevents us from including optimizations like top-k retrieval and impact ordering [1], that are induced by scoring and ranking, in our framework. Extending our method to more expressive query languages with advanced features like joins with arithmetic comparisons, nested queries, aggregation, top-k retrieval, etc. are part of the future work. Above query only represents the specific query "red ferrari". To represent all possible keyword search queries with two keywords, we use a "query template" shown below:

$$\begin{aligned}
Q_k[k_1, k_2](idf_1, idf_2, url, dId, len, tf_1, tf_2) : & -K(kId_1, k_1, idf_1), \\
& KD(kId_1, dId, tf_1), K(kId_2, k_2, idf_2), \\
& KD(kId_2, dId, tf_2), D(url, dId, len)
\end{aligned} \tag{2.4}$$

A query template is a conjunctive query whose variables are partitioned into bound and free variables. Here the head variables  $\{k_1, k_2\}$  are bound and other variables are free. Bound variables form input to the query. At run-time, each bound variable is replaced by a constant to obtain a "query instance" for the query template. For instance, the query instance  $Q_k["red", "ferrari"]$  for "red ferrari" is obtained by assigning  $k_1 = "red"$ ,  $k_2 = "ferrari"$  in the above query template. Such structures are also known "adorned views" in the literature [27]. We can now represent all keyword search queries with two keywords using the above query template. This allows us to represent a large number of query instances that share the general structure in a compact manner.

**Definition 2.1 *Corpus*:** a corpus  $C \sim \mathcal{C}$  is a database instance of schema  $\mathcal{C} = \{B_1, \dots, B_n\}$  defined by a set of base relations  $\{B_1, B_2, \dots, B_n\}$ .<sup>1</sup>

**Definition 2.2 *Query Template*:** a query template is a conjunctive query over a corpus of  $C \sim \mathcal{C}$ :

$$\begin{aligned}
Q[(V_b)_1, (V_b)_2, \dots, (V_b)_{n_b}]((V_f)_1, (V_f)_2, \dots, (V_f)_{n_f}) : & - \\
& g_1(x_1^1, x_2^1, \dots, x_{n_1}^1), g_2(x_1^2, x_2^2, \dots, x_{n_2}^2), \dots, g_k(x_1^n, x_2^n, \dots, x_{n_k}^n)
\end{aligned} \tag{2.5}$$

---

<sup>1</sup>Additional features like functional and inclusion dependencies can also be included in the definition of corpus. However, for simplicity in exposition, we don't include these extra features.

where  $V = \{(V_b)_1, (V_b)_2, \dots, (V_b)_{n_b}\}$  is the set of bound variables,  $V_f = \{(V_f)_1, (V_f)_2, \dots, (V_f)_{n_f}\}$  is the set of free variables,  $\forall i, g_i \in C$  and each  $x_j^i$  either lies in  $V_b \cup V_f$  or is a constant. A particular assignment of bound variables  $((V_b)_1, (V_b)_2, \dots, (V_b)_{n_b}) = ((v_b)_1, (v_b)_2, \dots, (v_b)_{n_b})$  to constant values produces a conjunctive query  $Q[(v_b)_1, (v_b)_2, \dots, (v_b)_{n_b}](V_f)$  called "query instance". A query template defines the set of all possible query instances which can be generated by substituting constant values in place of the bound variables  $V_b$ . We consider bag semantics for query execution. We use the notation  $\text{body}(Q)$ ,  $\text{boundvars}(Q)$  and  $\text{vars}(Q)$  to denote the sub-goals, bound variables and all variables of  $Q$  respectively.

Finally, a search system may be required to execute multiple query templates. Therefore, we formulate the system requirements as a query workload comprising of a set of query templates. It is important to consider the query workload as a whole rather than looking at each query individually while designing the system. This is because different queries may share index structures among them. For instance, keyword search queries of different lengths all use inverted index for query execution. Furthermore, different queries in the workload may have different weights or importance. For instance, a search query with 3 keywords may be more likely than a query with 100 keywords. In such a case, weight of a query template with length 3 would be higher than that of the length 100. We will use these weights to vary the importance of different query templates when we define the cost model in section-3.1. As a running example, we consider a keyword search system show in example-2.1 that executes document-to-keyword search queries along with the keyword-to-document search queries. A document-to-keyword search query is often used to retrieve the matching snippet for the documents in the rank list. We henceforth refer to this example as *keyword search example*.

**Example 2.1 Keyword search example** Consider a simplified corpus schema  $\mathcal{C} = \{K(k, d, p)\}$ . A record  $[k_1, d_1, p_1]$  in the relation  $K$  means that the keyword  $k_1$  appears in the document  $d_1$  at the position  $p_1$ . Now, consider a query workload that includes keyword-to-document and document-to-keyword search.

$$Q_k[k_1, k_2](d, p_1, p_2) : -K(k_1, d, p_1), K(k_2, d, p_2) \quad (2.6)$$

$$Q_d[d](k, p) : -K(k, d, p) \quad W = \{(w_k, Q_k), (w_d, Q_d)\} \quad (2.7)$$

Here  $w_k$  and  $w_d$  are the relative weights of the query templates  $Q_k$  and  $Q_d$  respectively. The weights are normalized, i.e.  $w_k + w_d = 1$ .

**Definition 2.3 Query Workload:** query workload for a system

$$W = \{(Q_1, w_1), (Q_2, w_2), \dots, (Q_k, w_k)\} \quad (2.8)$$

is a set of query templates with weights/frequencies  $w_i$ ,  $\sum_i w_i = 1$  associated to each query template based on their relative importance.

## 2.2 INDEX TABLES

We identify "index tables" as key building blocks of the physical design for an IR applications. These building block must serve two purposes. First, they should contain the information necessary to answer all query templates in the workload. For a database system, that supports general SQL queries, this role is played by its schema. However, unlike a database system, an IR application only needs to support a fixed query workload as defined in the previous section. Therefore, apart from containing the necessary information, they must be optimized w.r.t the specific workload. For instance, consider an IMDB like corpus  $C_{imdb} \sim \mathcal{C}_{imdb} = \{A(a, m), D(d, m)\}$ . Here the relation  $A(a, m)$  stores movies in column "m" and actors, that acted in that movie, in column "a". Similarly,  $D$  stores directors of a movie. Now say the query workload consists of only a single query template  $Q_{ad}[a](m, d) : -A(a, m), D(d, m)$ . Now, instead of storing the base relations  $A$  and  $D$ , we may store the view  $V(a, m, d) : -A(a, m), D(d, m)$ . The view  $V$  may speed up the query execution as the join movies is moved offline. Therefore, unlike a database system where the relations in the schema are the main source of data, basic building blocks for the design of an IR application must correspond to different views that can be used to support the workload.

An information retrieval query, as defined in definition-2.2, has the notion of input and output due to free and bound variables. For instance, the query template  $Q_k[k_1, k_2](d, p_1, p_2)$  from keyword search example looks-up documents and positions related to a given pair of keywords. The keywords  $(k_1, k_2)$  therefore form the input and the variables  $(d, p_1, p_2)$  form the output. Therefore, an inverted index is an efficient data structure to answer the query template  $Q_k$ . On the other hand, inverted index is extremely slow for answering the document-to-keyword query  $Q_d$  from the keyword search example. For  $Q_d$ , a "forward index" that maps a given document to the set of keywords present in it would be a more efficient data structure. Even though both the inverted index and the "forward index" contain the same information, given by the view  $V(k, d, p) : -K(k, d, p)$ , they differ in the way in which this information is organized. Therefore, the basic building block of an IR application, must also describe how the information contained in it is organised.

We formalize an index table as a function whose contents are the materialized result of

every query instance of a query template. The input to the function corresponds to the valuations of bound variables and the output corresponds to the valuations of free variables associated to it. For instance, the inverted index is defined as:  $INV[k](d, p) : -K(k, d, p)$ . Now, the input to an inverted index is the set of keywords that are present in at least one document. And the output, for a given keyword, corresponds to the list of valuations of variables  $(d, p)$  that satisfy  $K(k, d, p)$ . It can also be seen as a function  $f_{inv}$  over the relation  $K(k, d, p)$ . The index look-up is the function look-up over the input keyword "k", defined as:  $f_{inv}(x) = \pi_{d,p} \sigma_{k=x} K$ . Note, that this idea can be generalized to a tree of function rather than a single function. For example, consider a chain of functions  $f_{kd}(x) = f_{dp}(y; x)$  where  $f_{dp}(\cdot; x)$  is a function that takes input  $y$  and returns  $\pi_p \sigma_{k=x, d=y} K$ . However, for simplicity, we consider only the index tables that correspond to a single function.

**Definition 2.4 Index Table:** *an index table  $I$  is a function over the materialized result of a query template  $Q$  over the corpus  $C \sim \mathcal{C}$ . The input of  $I$  are the valuations of the bound variables  $V_b$  of the query  $Q$ . The index look-up  $I(v_b)$  on a set of constants  $v_b$  returns the result of the query instance of  $Q$  corresponding to the valuation  $V_b = v_b$ .*

These index look-ups can be implemented in many different ways. For instance, Lucene[22] is a popular implementation of inverted index. It implements the first column group, i.e. term dictionary, using the trie data structure. Index look-ups are performed by traversing the trie and obtaining the disk offset of the matching postings list. Different optimization are employed in Lucene to reduce the space and time complexity of the index look-up. For instance, the term dictionary is always cached in memory for fast access. Postings list is compressed for faster disk-IO. The definition-2.4 does not include these small implementation details. Instead, it provides an interface which defines how an index table might be used. We assume a black box implementation for an index table that may or may not include the optimizations employed in a library like Lucene. This implementation then defines a black box cost function, discussed in section-3.1, which we use to quantitative compare different index tables.

## 2.3 ANSWERING QUERIES USING INDEX TABLES

To execute a query template, we first store the indexes. Then we rewriting the query template using the indexes. And then we finally execute the rewriting conjunctive query. For instance, consider the keyword search query  $Q_k[k_1, k_2](d, p_1, p_2) : -K(k_1, d, p_1), K(k_2, d, p_2)$  and the inverted index  $I[k](d, p) : -K(k, d, p)$ . We can rewriting the query template  $Q_k$

using the inverted index as follows:

$$Q[k_1, k_2](d, p_1, p_2) : -I(k_1, d, p_1), I(k_2, d, p_2) \quad (2.9)$$

Here  $I(k_1, d, p_1)$  represents that we use the index  $I$  as a relation and assert the presence of the tuple  $[k_1, d, p_1]$  in it. We formalize this notion as an "index mapping" which describes how the variables of index should be mapped to the variables of the query. This example corresponds to the mapping:  $\{k : k_1, d : d, p : p_1\}$ . We require that an index mapping  $m : vars(I) \rightarrow vars(Q)$  must be an injective function, i.e.  $x \neq y \implies m(x) \neq m(y)$ . Furthermore, this mapping must define another injective mapping  $m_g : body(I) \rightarrow body(Q)$  where a sub-goal  $g(x_1, x_2, \dots, x_n) \in body(I)$  gets mapped to a sub-goal  $g(m(x_1), m(x_2), \dots, m(x_n)) \in body(Q)$ . In the above example, the sub-goal  $K(k, d, p)$  of the index  $I$  maps to the sub-goal  $K(k_1, d, p_1)$  under the mapping  $m$ . Because of these restrictions, all selections and joins in index are mapped to selection and joins within the query template.

Using this notion of index mapping, we now define the rewriting of a query template as a set of index mappings. For instance,  $\{I(k_1, d, p_1), I(k_2, d, p_2)\}$  is a rewriting of the query template  $Q_k$  using the mappings of index  $I$ . An index mapping can also be seen as an alias for the sub-goals present in the definition of the index. For instance,  $I(k_1, d, p_1)$  is an alias for the sub-goal  $K(k_1, d, p_1)$  as the two expressions represent the same assertion. By replacing each index mapping with the sub-goals of the underlying index we obtain the set of assertions implied by the rewriting. In the above example, we obtain the query template  $Q[k_1, k_2](d, p_1, p_2) : -K(k_1, d, p_1), K(k_2, d, p_2)$ , which is same as  $Q_k$ , by replacing the index mapping by its definition.

Note that the sub-goals may be repeated in case more than one index mappings share a sub-goal. For instance, consider an IMDB like corpus  $C_{imdb} = \{A(a, m), D(d, m)\}$ . Here the relation  $A(a, m)$  contain records  $[a, m]$  if the actor " $a$ " acted in a movie " $m$ ". Similarly, the relation  $D(d, m)$  stores the movies directed by a director. Now consider the query template  $Q_{aad}[a_1, a_2](m, d) : -A(a_1, m), A(a_2, m), D(d, m)$  and the index  $I[a](m, d)$ . The query template takes input as two actors  $a_1$  and  $a_2$  and returns the list of movies  $m$  in which they both acted and the directors of the movie. Now consider a rewriting  $R = \{I(a_1, m, d), I(a_2, m, d)\}$ . By replacing the index mappings with the definitions of the underlying index, we obtain the following query template of the rewriting:

$$Q[a_1, a_2](m, d) : -A(a_1, m), A(a_1, m), D(d, m), D(d, m) \quad (2.10)$$

Even though the sub-goal  $D(d, m)$  is repeated, it does not affect the result of the query.



We call a rewriting equivalent to the query if the execution of the rewriting yields the same result as the query. It is easy to see that a rewriting will be complete if the index mappings present in it map all the sub-goals of the query template at least once.

**Definition 2.5 Index Mapping:** *an index mapping  $m : \text{vars}(I) \rightarrow \text{vars}(Q)$  of index  $I$  within a query template  $Q$  is an injective mapping from  $\text{vars}(I)$  to  $\text{vars}(Q)$ . For an index mapping to be valid, there must be an injective mapping  $m_g : \text{body}(I) \rightarrow \text{body}(Q)$  that maps a goal  $g(x_1, x_2, \dots, x_n) \in \text{body}(I)$  to the goal  $g(m(x_1), m(x_2), \dots, m(x_n)) \in \text{body}(Q)$ . We use the notation  $m(g)$  to also refer to  $m_g(g)$ .*

**Definition 2.6 Rewriting:** *A rewriting  $R = \{m_1, m_2, \dots, m_n\}$  of a query template  $Q$ , over a set of indexes  $\mathcal{I}$ , is a set of valid index mappings  $m_i$  of indexes  $I_i \in \mathcal{I}$  within the query template  $Q$ . A rewriting is called complete if  $\cup_i m(g) = \text{body}(Q)$ , otherwise it is called partial.*

Having define the notion of a rewriting, we will describe how a rewriting may be executed. Index look-ups are the basic operations in our framework. The key consideration in execution of a rewriting is the order in which the index look-ups are performed. For instance, consider the query template  $Q_{aad}[a_1, a_2](d) : -A(a_1, m), A(a_2, m), D(d, m)$  and indexes  $I_a[a](m) : -A(a, m)$  and  $I_m[m](d) : -D(d, m)$  over the IMDB like schema  $C_{imdb} \sim \mathcal{C}_{imdb} = \{A(a, m), D(d, m)\}$ . Further consider the rewriting  $R_{aad} = \{I_a(a_1, m), I_a(a_2, m), I_m(m, d)\}$ . Now, for a given query instance, there are several ways in which the rewriting  $R_{aad}$  can be executed. We could first perform index look-up for  $I_a(a_1, m)$  to obtain the movies in which  $a_1$  acted. We could then perform index look-up on  $I_a(a_2, m)$  followed by  $I_m(m, d)$ . Note that we waited to perform index look-up on the column  $m$  of  $I_m$  until we obtained the set of movies in which both  $a_1$  and  $a_2$  acted. If we had performed look-up on the values of column  $m$  of  $I_m$  first, we would have to perform many more look-ups as we would not have known which movies will actually end up in the final query result.

We formalize the query plan of a rewriting as an order of index mappings in which the index look-ups will be performed. Given a plan, algorithm-2.1 describes a simple execution engine to execute it. For instance, consider the query plan  $P = [I_a(a_1, m), I_a(a_2, m), I_m(m, d)]$ . Figure-2.1 shows the steps involved performed by query execution engine for this the plan  $P$  on a toy dataset. It maintains a list of tuples corresponding to the current result at the end of each iteration in variable "result". "result" is initialized with columns associated to bound head variables of the query with a single tuples that lists the valuation  $v_b$  for each bound head variables. In the example shown in figure-2.1, bound head variables are initialized as  $a_1 = 2, a_2 = 3$ . Then the index mappings are processed in the order they specified by the plan  $P$ . In each step, algorithm-2.1 performs a natural join between the next index mapping and the result. This natural join is performed using index look-ups on the index table.

---

**Algorithm 2.1** Query Execution Engine
 

---

```

1: procedure EXECUTE( $Q, P, v_b$ )
2:    $result = \{columns = boundvars(Q), \quad values = v_b\}$ 
3:   for  $I_k \in P = [I_1, I_2, \dots, I_n]$  do
4:      $result = result$  (natural join)  $I_k$ 
5:   end for
6:   return  $result$ 
7: end procedure

```

---

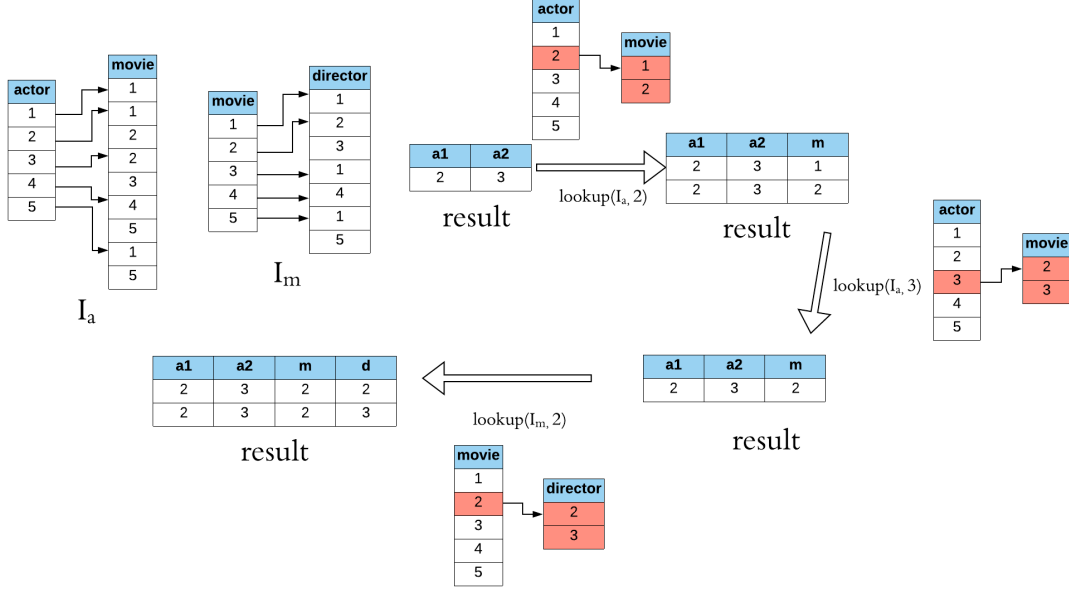


Figure 2.1: The execution of query instance  $Q[a_1 = 2, a_2 = 3](d) : -A(a_1, m), A(a_2, m), D(d, m)$  for the plan  $P = [I_a(a_1, m), I_a(a_2, m), I_m(m, d)]$  using the indexes  $I_a[a](m) : -A(a, m)$  and  $I_m[m](d) : -D(d, m)$ .

**Definition 2.7 Query Plan:** a query plan  $P$  for a rewriting  $R$  is a permutation of its index mappings. A plan is called *complete* if the rewriting associated to the set of index mappings in it is complete. If a plan  $P$  is a prefix of  $P'$ , then we define  $P \preceq P'$  to be true. A plan is *minimally complete* if all of its prefixes, except itself, are incomplete.

We can now formally define the physical index design of an IR application. Index design is composed of quantities that define how each query from the workload will be executed. It consists of: (i) the set of indexes to be stored, and (iii) a query plan for a complete rewriting for each query in the workload. For instance, a possible index design for the workload given in the keyword search example is as follows: (i) set of stored index  $I[k](d, p) : -K(k, d, p)$  and  $F[d](k, p) : -K(k, d, p)$ , and (ii) plan  $[I(k_1, d, p), I(k_2, d, p)]$  for  $Q_k$  and  $[F(d, k, p)]$  for  $Q_d$ . The index  $I$  is a representation of the inverted index. Its input column "k" is also

called "term-dictionary" in the literature [1]. The second column is called postings list for a given keyword. An index look up finds the keyword in the term dictionary and returns its postings list. The query plan  $[I(k_1, d, p), I(k_2, d, p)]$  therefore corresponds to performing index look-ups on each keyword then merging their postings lists.

**Definition 2.8 Index Design:** for a query workload  $W$  an index design  $D = (\mathcal{I}, \mathcal{P})$  contains: (i) the set of stored indexes  $\mathcal{I}$ , and (iii) the set of query plans  $\mathcal{P}$  with exactly one query plan,  $\mathcal{P}(Q)$  for each query template  $Q \in W$ . For an index design to be valid,  $\mathcal{I} = \{m.index \mid \exists Q \in W, m \in \mathcal{P}(Q)\}$ , where  $m.index$  is the underlying index for a given index mapping  $m$ . An index design is "complete" if each plan in  $\mathcal{P}$  is complete. We define the partial order  $\preceq$  as follows: given designs  $D = (\mathcal{I}, \mathcal{P})$  and  $D' = (\mathcal{I}', \mathcal{P}')$  if  $\forall Q \in W, \mathcal{P}(Q) \preceq \mathcal{P}'(Q)$ . We use the notation  $\text{complete}(D)$  to represent that a design  $D$  is complete. A design is minimally complete if all of its plans are minimally complete.

## CHAPTER 3: OPTIMIZATION FRAMEWORK

In this chapter, we layout the overall optimization framework within which we design our algorithm to obtain efficient index designs for a given query workload. We first formalize the set of candidate index design in section-3.1. It is obtained by first generating the set of candidate indexes and their index MCDs within the query workload. Candidate index designs are then all possible ways in which these candidate indexes and index MCDs can be used to execute the queries. If the number of candidate index MCDs is  $N$ , then there can be  $\frac{N!}{(N-k)!}$  possible query plans of size  $k$ . In practice,  $N$  will be much larger so  $N/2 \geq k$  is a reasonable assumption. Even in this case,  $\frac{N!}{(N-k)!} \geq (N/2)^k$ . The size of a query plan  $k$  is likely to be of the same order as the number of sub-goals in the query template. Since the space of candidate index designs can be exponential in the size of query workload, it is impractical even for small query workloads to exhaustively search this state space. Our strategy to deal with this key issue is as follows. We obtain a weight set cover [25] based relaxation of the problem in such a way that the optimal cost for the relaxed weighted set cover instance is either the upper bound or lower bound of the actual cost. We then use a well known greedy heuristic for this relaxed version to obtain a  $O(\log^2(n))$  ( $n$  = size of the workload) factor approximate solution in time linear in number of candidate index mappings. We then use the output of this heuristic algorithm within a branch-and-bound based strategy that prunes out sub-optimal index designs using these upper and lower bounds. The idea is to hopefully find reasonably good solutions using efficient heuristics and use them to prune out obviously bad solutions. Section-3.2 lays out the branch and bound based strategy and chapter-4 describes the weighted set cover based relaxation.

### 3.1 CANDIDATE INDEX DESIGNS AND COST MODEL

The set of candidate index designs is defined using the set of candidate indexes  $\mathbb{I}$  and the set of candidate index mappings corresponding to them  $\mathbb{M}$ . For instance, table-3.1 shows a list of candidate indexes and their index mappings for the workload of keyword-to-document and document-to-keyword queries defined in example-2.1. Here the inverted index  $INV[k](d, p) : -K(k, d, p)$  and the bi-word index  $BW[k_1, k_2](d, p_1, p_2) : -K(k_1, d, p_1), K(k_2, d, p_2)$  are both candidate index structures for the keyword search query template  $Q_k[k_1, k_2](d, p_1, p_2) : -K(k_1, d, p_1), K(k_2, d, p_2)$ . The candidate index mappings for the inverted index  $INV$  are  $\{k : k_1, d : d\}$  and  $\{k : k_2, d : d\}$ . Similarly, a possible index mapping for  $BW$  index is  $\{k_1 : k_1, k_2 : k_2, d : d\}$ . A candidate index design  $D = (\mathcal{I}, \mathcal{P})$  is now

Table 3.1: Set of candidate indexes and corresponding index MCDs for the running example-2.1

Candidate Indexes	Candidate Index MCDs
$INV[k](d, p) : -K(k, d, p)$	For $Q_k$ : $INV(k_1, d, p_1)$ , $INV(k_2, d, p_2)$ . For $Q_d$ : $INV(k, d, p)$ .
$FW[d](k, p) : -K(k, d, p)$	(same as $INV$ )
$BW[k_1, k_2](d, p_1, p_2) : -K(k_1, d, p_1), K(k_2, d, p_2)$	For $Q_k$ : $B(k_1, k_2, d, p_1, p_2)$ , $B(k_2, k_1, d, p_2, p_1)$ .

any index design which is built from the sets  $(\mathbb{I}, \mathbb{M})$

**Definition 3.1** *Space of candidate index designs: for a query workload  $W$ , the set of candidate indexes  $\mathbb{I}$  and the set of candidate index MCDs  $\mathbb{M}$ , the space of index designs  $\mathcal{D}(W, \mathbb{I}, \mathbb{M})$  contains all possible minimally complete index designs  $D = (\mathcal{I}, \mathcal{P})$  such that: (i)  $\mathcal{I} \subseteq \mathbb{I}$ , and (ii)  $\forall Q \in W, \mathcal{P}(Q) \subseteq \mathbb{M}$*

Our framework can be combined with any approach that generates the sets  $\mathbb{I}$  and  $\mathbb{M}$ . For instance, [28] describes an algorithm to generate the set of all possible useful views, for a given query workload, in a bottom-up manner. The algorithm starts with the initial candidate set of the most simple views that correspond to the base relations. For instance, the base relation  $K$  is a valid view in the query workload of our running example-2.1. It then generates novel views by merging already obtained views by applying all possible operations to obtain more and more complex views. For instance, the view  $V(k_1, k_2, d) : -K(k_1, d, p_1), K(k_2, d, p_2)$  is obtained by merging the view  $V(k, d, p) : -K(k, d, p)$  for the query template  $Q_k[k_1, k_2](d, p) : -K(k_1, d, p_1), K(k_2, d, p_2)$ . Now for each candidate view, we can enumerate the set of all possible column structures to generate indexes with that underlying view.

**Cost Model:** We consider two types of cost: (i) maintenance cost, and (ii) time cost. Maintenance cost includes the cost of storing, updating and maintaining an index. For instance, inverted index needs to be updated or recomputed every once in a while. It also needs to be stored in the disk. Time cost, on the other hand, is the estimation of query execution time. For inverted index based keyword search, query execution time would include reading and merging the postings list for each query keyword.

We assume black box functions following a specific function signatures for both kinds of cost. They can be implemented in any way as long as they satisfy the function signature.

$\text{maint}(I)$  gives maintenance cost for any index  $I$ . Similarly,  $\text{time}(P)$  gives the execution time for the plan  $P$ . Based on the query execution engine (algorithm-2.1), the  $\text{time}$  function has the following form:

$$\text{time}(P) = \sum_{m_i \in P} \text{time}(m_i|[m_1, m_2, \dots, m_{i-1}]) \quad (3.1)$$

Here  $P = [m_1, m_2, \dots, m_n]$ . The terms in the above expression corresponds to the iterations in algorithm-2.1.  $i^{\text{th}}$  term  $\text{time}(m_i|[m_1, m_2, \dots, m_{i-1}])$  corresponds to performing natural join of the index mapping  $m_i$  with the intermediate result. For instance, consider the query plan  $P = [I_a(a_1, m), I_a(a_2, m), I_m(m, d)]$  for the query  $Q_{aad}$  shown in figure-2.1. The cost of first iteration  $\text{time}(I_a(a_1, m)|[])$  corresponds to performing the index look-up on the index  $I_a$  for the given input value  $a_1$ . Similarly, second stage cost  $\text{time}(I_a(a_2, m)|[I_a(a_1, m)])$  equals the time it takes to perform index look-up on  $I_a$  with given value  $a_2$ . The final term,  $\text{time}(I_m(m, d)|[I_a(a_1, m), I_a(a_2, m)])$ , includes the cost to perform index look-ups for all the movies  $m$  in which both  $a_1$  and  $a_2$  acted.

**Definition 3.2 Cost Model:** for an index  $I$ ,  $\text{maint}(I)$  represents the storage and maintenance cost associated to it. For a query plan  $P = [m_1, \dots, m_n]$  with "n" stages, cost estimate of its query execution time is defined as:

$$\text{time}(P) = \sum_{m_i \in P} \text{time}(m_i|[m_1, m_2, \dots, m_{i-1}]) \quad (3.2)$$

where  $\text{time}(m_i|[m_1, m_2, \dots, m_{i-1}])$  represents the time required to merge the index MCD  $m_i$  with the result of the previous index MCDs  $[m_1, m_2, \dots, m_{i-1}]$  in the line-4 of the algorithm-2.1. Now, cost of a design  $D = (\mathcal{I}, \mathcal{P})$  is given as:

$$\text{cost}(D) = \left( \sum_{I \in \mathcal{I}} \text{maint}(I) \right) + \left( \sum_{Q \in W} w_Q * \text{time}(\mathcal{P}(Q)) \right) \quad (3.3)$$

Finally, the problem of obtaining an efficient index design for a given query workload can be stated as follows.

**Definition 3.3 IDP[ $W, \mathbb{I}, \mathbb{M}, \text{cost}$ ]:** the index design problem for a space of index designs  $\mathcal{D}(W, \mathbb{I}, \mathbb{M})$ , the cost model  $\text{cost} = (\text{maint}, \text{time})$  is to find least cost complete design  $\mathcal{D} \in \mathcal{D}(W, \mathbb{I}, \mathbb{M})$

### 3.2 BRANCH-AND-BOUND SEARCH

The space of candidate index design described in previous section can be exponential in the size of the query workload  $|W|$ . If  $N = |\mathbb{M}|$  is number of candidate index mappings for a single query template, then there are  $\frac{N!}{(N-k)!}$  ways to construct a query plan of length  $k$  out of it. In most scenarios,  $N$  would be much larger than  $k$ . Assuming  $N/2 \geq k$ , there are  $\frac{N!}{(N-k)!} \geq (N/2)^k$  possible query plans. Moreover,  $N$  might itself be exponential in  $|W|$  as there are  $2^n$  different subsets of  $body(Q)$  ( $n = |body(Q)|$ ) which might potentially form a unique index mapping. An exhaustive search of such a large space of solutions is impractical even for small query workloads. Therefore, we design efficient heuristics that give upper and lower bounds to the actual cost in polynomial time in chapter-4. However, since there are no guarantees on the performance of these heuristics, we use them as subroutines inside a branch and bound based algorithm. The branch-and-bound algorithm can provably find the optimal solution but it may take exponential time in the process.

**Branching rule:** branching rule is one of the main components for any branch-and-bound based algorithm [24]. A branching rule provides a way to partition the search space into disjoint sub-spaces. To describe our branching rule, we will first describe an alternate way to represent a valid index design  $D \in \mathcal{D}(W, \mathbb{I}, \mathbb{M})$  defined in definition-3.1. Consider the set  $S(\mathbb{M}) = \mathbb{M} \times \{1, 2, \dots, k\}$  where  $k$  is some fixed number bounding the size of any complete plan we may want to find. Now, every design  $D$  in the solution space, can be seen as a subset of  $S(\mathbb{M})$  as follows. For any given  $D = (\mathcal{I}, \mathcal{P})$ , let the set  $S(D) = \{(m_i, i) | \exists Q \in W, \mathcal{P}(Q) = [m_1, m_2, \dots, m_i, \dots, m_n]\}$ . For instance, the design  $D = (\{INV[k_1](d, p)\}, \{Q_k : [INV(k_1, d, p), INV(k_2, d, p)], Q_d : []\})$  for the keyword search example, corresponds to  $S(D) = \{(INV(k_1, d, p), 1), (INV(k_2, d, p), 2)\}$ . The set  $S(D)$  is essentially a unique set representation of the query plans such that,  $\forall D, D' \in \mathcal{D}(W, \mathbb{I}, \mathbb{M}), D \neq D' \implies S(D) \neq S(D')$ . Furthermore, note that  $\forall D \in \mathcal{D}(W, \mathbb{I}, \mathbb{M}), S(D) \subseteq S(\mathbb{M})$ . We can use this representation to obtain a compact represent the space of index designs and a way to quickly partition it into disjoint sub-spaces. To do so, we arrange the elements of  $S(\mathbb{M})$  into a sequence  $L$ . For example, consider the following set of candidate index and index MCDs for the keyword search example:

$$\mathbb{I} = \{I[k](d, p) : -K(k, d, p), F[d](k, p) : -K(k, d, p)\} \quad (3.4)$$

$$\mathbb{M} = \{I(k_1, d, p), I(k_2, d, p), I(k, d, p), F(d, k_1, p), F(d, k_2, p), F(d, k, p)\} \quad (3.5)$$

Also, say we arrange the elements of  $\mathbb{S}(M)$  into the following sequence  $L$ :

$$\begin{aligned} L = & [(I(k_1, d, p), 1), (I(k_2, d, p), 1), (I(k, d, p), 1), (F(d, k_1, p), 1), \\ & (F(d, k_2, p), 1), (F(d, k, p), 1), (I(k_1, d, p), 2), (I(k_2, d, p), 2), \\ & (I(k, d, p), 2), (F(d, k_1, p), 2), (F(d, k_2, p), 2), (F(k, d, p), 2)] \end{aligned} \quad (3.6)$$

Now, every possible index design can be generated by iterating over this sequence from left to right and making a binary decision to either include an element like  $(m, t)$  in our design or not. Including element  $(m, t)$  means that the index MCD  $m$  appears at position  $t$  of the query plan of the query template underlying  $m$ . We can now define a sub-space of index designs using a pair  $(S_n, n)$  where  $n \in \{0, 1, 2, \dots, |L|\}$  and  $S_n$  is a subset of first  $n$  elements of  $L$ . For instance, consider the pair  $(\{I(k_1, d, p), 1), F(d, k_1, p), 1\}, 4)$ . It represents all the designs which can be obtained by extending the design  $D = (\{I, F\}, \{Q_k : [I(k_1, d, p), 1], Q_d : [F(d, k_1, p), 1]\})$  using last  $|L| - 4 = 8$  elements of  $L$ . The pair  $(\phi, 0)$  corresponds to the space of all possible index designs containing query plans of length  $\leq k$ . And the pair  $(S(D), |L|)$  represents the sub-space containing the single design  $D$ .

**Definition 3.4** *Sub-spaces of candidate index designs: for any index design  $D$ , define  $S(D) = \{(m_i, i) | \exists Q \in W, \mathcal{P}(Q) = [m_1, m_2, \dots, m_i, \dots, m_n]\}$ . For a space of candidate index designs  $\mathcal{D}(W, \mathbb{I}, \mathbb{M})$  and a sequence  $L$  over elements of the set  $\mathbb{M} \times \{1, 2, \dots, k\}$ , the subspace  $\mathcal{S}(S_n, n) \subseteq \mathcal{D}(W, \mathbb{I}, \mathbb{M})$ , where  $S_n$  is a subset of first  $n$  elements of  $L$  is the set of all designs  $D \in \mathcal{D}(W, \mathbb{I}, \mathbb{M})$  such that: (i)  $S(D) = S_n \cup S'_n$  where  $S'_n$  is a subset of last  $|\mathbb{M} \times \{1, 2, \dots, k\}| - n$  elements of  $L$ , (ii)  $D$  contains plans of length at most  $k$ .*

Now, we can partition or "branch" a sub-space  $\mathcal{S}(S_n, n)$  into two disjoint sub-spaces  $\mathcal{S}(S_n \cup \{L_n\}, n+1)$  and  $\mathcal{S}(S_n, n+1)$  (using zero based indexing of  $L$ ). Since no two index MCDs can be present at the same positions in a query plan, any subspace  $\mathcal{S}(S_n, n)$  such that,  $(m, t), (m', t) \in S_n$  and  $m' \neq m$ , is empty. We use the notation  $branch(S_n, n)$  to denote a sub-routine that branches the sub-space  $\mathcal{S}(S_n, n)$  as defined above. Although, in principle, the definition of sub-space works for any sequence  $L$ , we only consider the sequences generated as follows. Order the elements of  $\mathbb{M}$  into a sequence  $L_M = [m_1, m_2, \dots, m_n]$ . Now define the sequence  $L$  as follows:

$$\begin{aligned} L = & [(m_1, 1), (m_2, 1), \dots, (m_n, 1), (m_1, 2), (m_2, 2), \dots, (m_n, 2), \\ & \dots, (m_1, k), (m_2, k), \dots, (m_n, k)] \end{aligned} \quad (3.7)$$

This way of sequence generation is important for the algorithm described in chapter-4 to work. It also intuitively makes sense. The above sequence forces the generation of  $t^{th}$  index



MCD in the query plan for a query  $Q$  before the generation of  $(t + 1)^{th}$  index MCD of the query plan of some other query  $Q'$ . This is because if the set  $S_n$ , in the sub-space  $\mathcal{S}(S_n, n)$ , does not contain any element of form  $(m, t)$  (with the underlying query template  $Q$  for  $m$ ) and  $L_n$  is of form  $(m', t + 1)$  (with the underlying query template  $Q$  for  $m$ ) then there is no way to construct a valid index design within  $\mathcal{S}(S_n, n)$ . Therefore,  $\mathcal{S}(S_n, n)$  is empty in such a case. This sequence therefore makes sure that no query template is "starved" during the generation process. Algorithm-3.1 now formalizes the way in which a given sub-space is branched into partitions of it.

---

**Algorithm 3.1** Branching rule

---

```

1: procedure BRANCH( $\mathcal{S}(S_n, n)$ )
2:   Generate  $L$  as per equation-3.7
3:    $neighbors = \phi$ 
4:   Let  $D_n = (\mathcal{I}_n, \mathcal{P}_n)$  be such that  $S(D_n) = S_n$ 
5:   for  $k \in \{n, n + 1, \dots, |L|\}$  do
6:     if  $\exists D, S(D) = S_n \cup \{L_k\}$  then
7:       if  $\forall Q, \mathcal{P}(Q)$  is complete  $\vee L_k = (m, t), t \leq 1 + |\mathcal{P}_n(Q)|$  then
8:          $neighbors = neighbors \cup \{\mathcal{S}(S_n \cup \{L_k\}, k + 1)\}$ 
9:       end if
10:    end if
11:  end for
12:  return  $neighbors$ 
13: end procedure

```

---

Given a sub-space  $\mathcal{S}(S_n, n)$ , algorithm-3.1 enumerates all choices of adding next element with position  $\geq n$  to  $S_n$ . Because of the design of the sequence  $L$ , if  $\nexists D, S(D) = S_n$  then the sub-space represented by  $\mathcal{S}(S_n, n)$ . Following lemma formalizes this idea.

**Lemma 3.1** *Provided we start with the sub-space  $\mathcal{S}(\phi, 0)$ , sub-space generated by the algorithm-3.1 takes form:*

$$\begin{aligned} \mathcal{S}(S_n, n) = \{D \mid D = (\mathcal{I}, \mathcal{P}), D_n \in \mathcal{D}(W, \mathbb{I}, \mathbb{M}), \\ S(D_n) = S_n, D \succeq D_n, \forall Q \in W, |\mathcal{P}(Q)| \leq k\} \end{aligned} \quad (3.8)$$

Based on [24], algorithm-3.2 describes the general structure of our branch-and-bound state space search algorithm. It assumes access to the upper and lower bounding heuristics  $UB(\mathcal{S}(S_n, n))$  and  $LB(\mathcal{S}(S_n, n))$  that provides upper and lower bounds respectively on the least cost complete designs  $D \in \mathcal{S}(S_n, n)$ .  $UB(\mathcal{S}(S_n, n))$  additionally provides a design with the cost less than or equal to the upper bound. In chapter-4, we discuss the implementation of these functions. Using these functions, algorithm-3.2 maintains least upper bound  $Lub$

and greatest lower bound  $Glb$  on the optimal cost. It also maintain the set *active* which represents the designs that have been discovered but not yet explored.

In every iteration it picks a sub-space to be explored in line-6. Now, the main mechanism through which it avoids an exhaustive search is at the line-8 of the algorithm. If the lower bound on least cost of a design in the given sub-space is greater than the best solution known till now, i.e.  $LB(\mathcal{S}(S_n, n)) \geq Lub$ , then there is no point in exploring further designs within that subspace. A sub-space for which  $LB(\mathcal{S}(S_n, n)) < Lub$ ,  $branch(\mathcal{S}(S_n, n))$  provides a set of sub-spaces that forms a partition of  $\mathcal{S}(S_n, n)$ .

---

**Algorithm 3.2** Branch and bound algorithm[24]

---

```

1: procedure BRANCHANDBOUND( $IDP[W, (\mathbb{I}, \mathbb{M}), cost, LB, UB, pick, expand, \epsilon]$ )
2:   Generate  $L$  as per equation-3.7
3:    $Glb = LB(\mathcal{S}(\phi, 0))$ 
4:    $Lub, D^* = UB(\mathcal{S}(\phi, 0))$ 
5:    $active = \{\mathcal{S}(\phi, 0)\}$ 
6:   while  $Lub - Glb > \epsilon$  and  $active \neq \phi$  do
7:      $\mathcal{S}(S_n, n) = pick(active)$ 
8:     if  $LB(\mathcal{S}(S_n, n)) < Lub$  then
9:       for all  $(\mathcal{S}(S_{n'}, n'))$  in  $branch(\mathcal{S}(S_n, n))$  do
10:         $active = active \cup \{\mathcal{S}(S_{n'}, n')\}$ 
11:         $ub', D'_{ub} = UB(\mathcal{S}(S_{n'}, n'))$ 
12:        if  $ub' < Lub$  then
13:           $Lub = ub'$ 
14:           $D^* = D'_{ub}$ 
15:        end if
16:      end for
17:    end if
18:     $active = active \setminus \{\mathcal{S}(S_n, n)\}$ 
19:  end while
20:  return  $D^*$ 
21: end procedure

```

---

**Theorem 3.1** [24] *Algorithm-3.2 returns an index design with cost  $\leq OPT + \epsilon$  where  $OPT$  represents the optimal cost assuming the following holds: (i)  $branch(\mathcal{S}(S_n, n))$  returns a partitions of  $\mathcal{S}(S_n, n)$ , (ii)  $LB(\mathcal{S}(S_n, n))$  and  $UB(\mathcal{S}(S_n, n))$  provides the lower and upper bound on any complete design  $D \in \mathcal{S}(S_n, n)$  respectively, and (iii) if  $\exists D_n, S(D_n) = S_n \wedge complete(D_n)$ , then  $UB(\mathcal{S}(S_n, n))$  provides the cost of  $D_n$ .*

**Sub-problem selection rule:** the "pick" in the line-5 of algorithm-3.2 plays an important role in directing the search. There are several choices for "pick" function that make

sense.

1. Pick the sub-space  $\mathcal{S}(S_n, n) \in \text{active}$  which minimizes  $LB(\mathcal{S}(S_n, n))$ . This is the  $A^*$ -search strategy [29]. If we only don't have access to  $LB$  sub-routine, then this strategy is provably best among all other strategies for *pick* function [29]. However, in case the bounds provided by  $LB$  are of low quality, then this strategy will not work very well.
2. Pick the sub-space  $\mathcal{S}(S_n, n) \in \text{active}$  which minimizes  $UB(\mathcal{S}(S_n, n))$ . This strategy is useful when the bounds provided by  $UB$  are close to actual cost.
3. Pick the sub-space  $\mathcal{S}(S_n, n) \in \text{active}$  with highest values of  $n$ . Higher the values of  $pos$ , smaller the size of the sub-space represented by  $\mathcal{S}(S_n, n)$ . As the sub-space of index designs become smaller, the gap between lower and upper bounds on the optimal cost reduces. Therefore, the intuition behind this strategy is to generate complete designs as quickly as possible with the hope that  $Lub$  will reduce significantly due to better estimates of  $UB$  cost.

In chapter-6, we perform experiments comparing each of the three options for the *pick* function.

## CHAPTER 4: WEIGHTED SET COVER BASED RELAXATION

Bounding functions,  $UB$  and  $LB$ , play a crucial role in the efficiency of algorithm-3.2. For instance, if we use trivial bounds like  $LB(\mathcal{S}(S_n, n)) = 0, UB(\mathcal{S}(S_n, n)) = \infty$ , for all designs  $D$ , then the algorithm-3.2 degenerates to an exhaustive search. In this section, we will describe a weighted set cover based relaxation of the index design problem. The universe of elements to be covered is the set of all sub-goals within a query. Each index mapping covers a subset of this universe. A complete rewriting is analogous to a cover of the elements of the universe using the subsets defined by index mappings. The only difference between the set cover formulation and the index design problem (definition-3.3) is that the cost of an index design is computed in terms of query plans and not rewriting. Therefore, unlike weight set cover problem, the order in which the index mappings are considered matter. In section-4.1, we describe a "context independent" cost models that allow us to assign weights to each index mapping, independent of the order of other index mappings in the query plan. The weight are assigned in such a way that the weight cost of the set cover either upper bounds or lower bounds the cost of any query plan. In section-4.2, using the context independent cost, we obtain a weighted set cover instance for any given instance of index design problem. We obtain two set cover instance corresponding to  $UB$  and  $LB$ . Finally, we describe a greedy algorithm that approximates the optimal cost, for the above weighted set cover instances, within a factor of  $O(\log^2(|W|))$ , where  $|W|$  is the size of the query workload.

### 4.1 CONTEXT INDEPENDENT COST

The index design problem can be written as the following optimization problem:

$$\mathcal{I}^*, \mathcal{P}^* = \arg \min_{D=(\mathcal{I}, \mathcal{P}), D \in \mathcal{S}(S_n, n)} \sum_{I \in \mathcal{I}} \text{maint}(I) + \sum_{Q \in W} w_Q \sum_{m_i \in \mathcal{P}(Q)=[m_k, m_{k+1}, \dots, m_l]} \text{time}(m_i | [m_1, m_2, \dots, m_{i-1}]) \quad (4.1)$$

Let  $D_n = (\mathcal{I}_n, \mathcal{P}_n)$  be such that  $S(D_n) = S_n$ . To obtain the weighted set cover based relaxation, we modify the above optimization problem as follows:

$$\mathcal{I}^*, \mathcal{P}^* = \arg \min_{\mathcal{I}, \mathcal{P}} \text{cost}(D_n) + \sum_{I \in \mathcal{I} \setminus \mathcal{I}_n} \text{maint}(I) + \sum_{Q \in W} \sum_{m \in \mathcal{P}(Q) \setminus \mathcal{P}_n(Q)} \text{time}_{CI}[\mathcal{P}_n(Q)](m) \quad (4.2)$$

Here,  $\mathcal{P}(Q) \setminus \mathcal{P}_n(Q) = [m_j, m_{j+1}, \dots, m_l]$  denotes the extra index mappings present in  $\mathcal{P}(Q)$

as compared to  $\mathcal{P}_n(Q)$ . Note that this quantity is well defined as  $D \succeq D_n$  due to lemma-3.1. In the previous equation, the term  $time(m_i|[m_1, m_2, \dots, m_{i-1}])$  represents the contribution of the index mapping  $m_i$  to the final cost as a function of its "context" i.e. the part of query plan which came before it. We simplify this term by replacing it with  $time_{CI}[\mathcal{P}_n(Q)](m)$  in the second equation. The term  $time_{CI}[\mathcal{P}_n(Q)](m)$  represents a "context independent" approximate cost contribution of  $m$  for all query plans  $\mathcal{P}(Q)$  extending  $\mathcal{P}_n(Q)$ . It is called "context independent" because its value does not depend on the index mapping that come after the ones in  $\mathcal{P}_n(Q)$ .

We design the function  $time_{CI}[\mathcal{P}_n(Q)](m)$  such that it either upper bounds or lower bounds the *context dependent* term  $time(m_i|[m_1, m_2, \dots, m_{i-1}])$ . To achieve this, we make certain assumptions on the cost model. Note that because of the design of query execution engine (algorithm-2.1), the term  $time(m|P)$  is expected to be smaller for longer query plans  $P$ . For instance, compare the plan  $[I_a(a, m), I_m(m, d)]$  with  $[I_m(m, d)]$  for the query template  $Q[a](m, d) : -A(a, m), D(d, m)$  and indexes  $I_a[a](m) : -A(a, m)$ ,  $I_a = m[m](d) : -D(m, d)$ . The cost contribution of  $I_m(m, d)$  in the first plan is  $time(I_m(m, d)|[I_a(a, m)])$  which corresponds to performing index look-ups for movies in which the given actor  $a$  acted. This value is expected to be much smaller than the cost contribution if second plan  $time(I_m(m, d)|[])$  which corresponds performing index look-ups for all movies in the corpus. Although the *time* function may be complicated and dependent on implementation, it is reasonable to assume that  $time(m|P)$  is smaller with longer histories  $P$ . This idea is formalized in the following assumption:

#### Assumption 4.1

$$P \succeq P' \implies time(m|P) \leq time(m|P') \quad (4.3)$$

Using the above assumption, the definition  $time_{CI}[\mathcal{P}_n(Q)](m) = time(m|\mathcal{P}_n(Q))$  provides an upper bound on the contribution of  $m$  when included in  $\mathcal{P}(Q) \setminus \mathcal{P}_n(Q)$  for any  $\mathcal{P}(Q) \succeq \mathcal{P}_n(Q)$ . We use this context independent cost to define a weighted set cover based relaxation in section-4.2. Next, we obtain a lower bound on the context dependent term  $time(m_i|[m_1, m_2, \dots, m_{i-1}])$ . Consider the term  $time(m|P)$  where  $P$  is a complete query plan. For instance,  $P = [I_a(a, m), I_m(m, d)]$  is a complete query plan in the above example. Now the value  $time(m|P)$  is expected to be as low as it can get because we already know the exact values of  $m$  which will make into the final result of the query. For instance,  $time(I_m(m, d)|P)$  corresponds to looking up the movies in the given actor acted and some director directed it. It is therefore reasonable to assume that  $time(m|P)$  is a lower bound on every term  $time(m|P')$  if the plan  $P$  is complete. This assumption can be formalized as:

**Assumption 4.2** *If  $P$  is complete, then  $\forall m, P' : \text{time}(m|P) \leq \text{time}(m|P')$*

Now, using the above assumption, we can obtain the lower bounding context dependent cost given as:  $\text{time}_{CI}[\mathcal{P}_n(Q)](m) = \text{time}(m|P_c)$  for some complete plan  $P_c$ . We will use this lower bounding context independent cost to define a lower bounding set cover instance in section-4.2 which will then be used as the sub-routine  $LB$  in algorithm-3.2.

**Definition 4.1** *Context Independent Time Cost: for a given initial design  $D_0 = (\mathcal{I}_0, \mathcal{P}_0)$ , candidate indexes  $\mathbb{I}$ , candidate index mappings  $\mathbb{M}$  and cost model "time", the under and over estimated context independent cost is defined as follows.*

1. *Over-estimated time cost is defined as:*

$$\text{time}_{oe}[\mathcal{P}_0(Q)](m) = \text{time}(m|\mathcal{P}_0(Q)) \quad (4.4)$$

2. *Under-estimated time cost is defined as:*

$$\text{time}_{ue}[\mathcal{P}_0(Q)](m) = \text{time}(m|P_Q^c) \quad (4.5)$$

where  $P_Q^c$  is any complete query plan of a complete rewriting of the query template  $Q$ .

## 4.2 TWO-LEVEL WEIGHTED SET COVER PROBLEM

In this section, we use the context independent cost to obtain a weighted set cover based relaxation of the index design problem. The key idea behind the relaxation is to simulate the decision of using an index mapping, as picking an available set in the set cover instance. Consider the following instance of the index design problem for the candidate indexes in table-3.1 for the keyword search example.

**Example 4.1** *The initial design  $D_0 = (\mathcal{I}_0, \mathcal{P}_0)$  where: (i)  $\mathcal{I}_0 = \{INV[k](d, p) : -K(k, d, p)\}$ , (ii)  $\mathcal{P}_0 = \{Q_k : [INV(k_1, d, p)], Q_d : []\}$ . Let the maintenance costs of the indexes  $INV, FW$ , and  $BW$  be 5000, 5100 and 50,000 respectively. Also, let the  $\text{cost}(D_0) = 5600$  and the context independent cost estimates for the index  $MCDs$  as per table-4.1.*

In the above example, we are interested in finding a least cost complete design  $D = (\mathcal{I}, \mathcal{P}) \succeq D_0$ . Here the candidate indexes and their mappings,  $(\mathbb{I}, \mathbb{M})$ , are shown in table-3.1. This problem is formulated as a set cover instance as follows. We first define the set of all sub-goals in the workload as universe of elements  $U$ , which  $D$  must cover. In this example,

Table 4.1: Context independent time cost estimates for the example-4.1. Cost estimates  $time_{ue}(m)$  and  $time_{oe}(m)$  are given for their corresponding query plan in the initial design  $D_0$  of example-4.1.

index MCD, $m$	$time_{ue}(m)$	$time_{oe}(m)$
$INV(k_2, d, p_2)$	500	500
$INV(k_1, d, p_1)$	0	0
$INV(k, d, p)$	4400	100,000
$FW(d, k, p)$	500	500
$FW(d, k_1, p)$	3600	100,000
$FW(d, k_2, p)$	3600	100,000
$BW(k_1, k_2, d, p_1, p_2)$	200	200

$U = \{K(k_1, d, p_1), K(k_2, d, p_2), K(k, d, p)\}$ . Next, we identify the subset of elements already covered  $U_c$ . In this example, the set  $U_c = \{K(k_1, d, p_1)\}$ .

Next, we define the set of available sets  $A$ , which we can pick from, in order to cover the uncovered elements  $U \setminus U_c$ . We partition the set  $A$  into  $A[I]$  for each index  $I \in \mathbb{I}$ , where  $A[i]$  represents the index mappings of  $I$ .  $A[I]$  is defined separately for the indexes  $I \in \mathcal{I}_0$  and  $I \in \mathbb{I} \setminus \mathcal{I}_0$ . This is because using an index mapping of an index  $I \in \mathcal{I}_0$  does not incur maintenance cost where using non-zero index mappings of indexes  $I \in \mathbb{I} \setminus \mathcal{I}_0$  incurs a one-time maintenance cost. For the first case,  $I \in \mathcal{I}_0$ , we define an available set  $a \in A[I]$  for every index mapping  $m$  of the index  $I$ . In this example,  $INV$  is the only index in  $\mathcal{I}_0$ . Now, for the index mapping  $m = INV(k_2, d, p_2)$  we define the available set  $a = \{K(k_2, d, p_2)\}$  containing the elements in  $U \setminus U_c$  covered by  $m$ . The weight for this set is defined as  $w(a) = time_{CI}[\mathcal{P}_0(Q_k)](m) = 500$ , where  $CI \in \{oe', ue'\}$ . Intuitively, picking the set "a" corresponds to including the index MCD  $m$  in the plan  $\mathcal{P}(Q_k)$  with its cost contribution being  $w(a)$ . However, the same strategy of defining the available sets does not work for the indexes  $I \in \mathbb{I} \setminus \mathcal{I}_0$  because they also incur a one-time cost of  $maint(I)$  to store/maintain the index. For the second case of  $I \in \mathbb{I} \setminus \mathcal{I}_0$ , we define an available set  $a \in A[I]$  corresponding to the decision of storing  $I$  and including a *subset* of its candidate index mappings within the plans  $\mathcal{P}$ . For instance, consider the subset of candidate index MCDs  $M_{fw} = \{FW(d, k, p), FW(d, k_2, p_2)\}$  for the forward index  $FW$ . The available set for  $M_{fw}$  contains the uncovered query elements covered by the index mappings in  $M_{fw}$ :  $a = \{K(k, d, p), K(k_2, d, p_2)\}$ . Picking the set  $a$  corresponds to the decision of storing the index  $FW$  and using its index mapping in  $M_{fw}$  in the final complete design  $D$ . As a result, the weight for this set is defined as:  $w(a) = 5100 + 500 + 3600$  for the lower bounding set cover instance, and  $w(a) = 5100 + 500 + 100,000$  for the upper bounding instance. We define

an available set in this manner for every possible subset of candidate index mappings for a given index. The intuition is that for an optimal complete design  $D^*$  there exists an optimal subset of index MCDs for each index  $I \in \mathbb{I} \setminus \mathcal{I}_0$  which will eventually be included in the plans of  $D^*$ . Including all such choices in the set of available sets  $A[I]$  ensures that there will be a set cover associated to the choices made by  $D^*$ . The overall collection of available sets  $A = \cup_{I \in \mathbb{I}} A[I]$ .

Finally, the weighted set cover problem can be formulated as the following optimization problem:

$$A^* = \arg \min_{A \subseteq \mathcal{A}} \sum_{a \in A} w(a), \quad s.t. \cup_{a \in A} a \supseteq U \setminus U_c \quad (4.6)$$

The corresponding design  $D^* = (\mathcal{I}^*, \mathcal{P}^*)$  is then obtained by: (i) including all indexes whose at least one mapping is used to construct a subset  $a \in A^*$ , (ii) including all index mappings used in some  $a \in A^*$  in the plans  $\mathcal{P}^*$  in any order after the initial plans  $\mathcal{P}$ .

**Definition 4.2**  $WSC[W, \mathbb{I}, \mathbb{M}, cost, time_{CI}[D_0], D_0]$ : for an instance of index design problem, starting from the initial design  $D_0 = (\mathcal{I}_0, \mathcal{P}_0)$  and a context independent time cost " $time_{CI}[D_0]$ " (where  $CI \in \{"oe", "ue"\}$ ), define the following:

- $U = \cup_{Q \in W} \{g | g \in body(Q)\}$  is the universe of elements to be covered
- $U_c = \cup_{m \in \mathcal{P}_0(Q), Q \in W} m(body(m.I))$  is the subset of element of  $U$  already covered by  $D_0$ . Here  $m.I$  is the underlying index of the mapping  $m$  and  $m(body(m.I))$  represents the set of goals within the body of  $Q$  that are mapped by the goal in the body of  $m.I$ . Therefore,  $U \setminus U_c$  is the set of extra elements to be covered.
- Define the set of available sets  $A[I]$ , for each index  $I \in \mathbb{I}$ , and weight function  $w : A \rightarrow \mathbb{R}^+$  as follows:
  1. For all  $m \in \mathbb{M}$  of index  $I \in \mathcal{I}_0$  within query  $Q \in W$ ,  $a = m(body(m.I)) \in A[I]$  and  $w(a) = time_{CI}[\mathcal{P}_0(Q)](m)$
  2. For all indexes  $I \in \mathbb{I} \setminus \mathcal{I}_0$  and subsets of their candidate index mappings  $M_I$ , the available set  $a$  is given by,

$$a = \cup_{m \in M_I} m(body(m.I)) \in A[I] \quad (4.7)$$



and, its weight  $w(a)$  is defined as

$$w(a) = \text{maint}(I) + \sum_{m \in M_I} \text{time}_{CI}[\mathcal{P}_0(m.Q)](m) \quad (4.8)$$

where  $m.Q$  is the query template underlying  $m$ .

Now the set of available sets is:  $A = \cup_{I \in \mathbb{I}} A[I]$ . The weighted set cover problem  $WSC[W, \mathbb{I}, \mathbb{M}, \text{cost}, \text{time}_{CI}[D_0], D_0]$  is then defined as:

$$A^* = \arg \min_{A' \subseteq A} \text{cost}_{CI}(D_0) + \sum_{a \in A'} w(a), \quad \cup_{a \in A'} a \supseteq U \setminus U_c \quad (4.9)$$

The above definition defines two weighted set cover instance. First one uses  $\text{time}_{ue}$ , for each index mapping, to assign the weights to the available sets. The second one uses  $\text{time}_{oe}$ . Theorem-4.1 now proves that the optimal cost for the two set cover instances provides upper and lower bounds for the original problem and therefore can be used as the sub-routines  $UB$  and  $LB$  in the algorithm-3.2.

**Theorem 4.1** *Let  $D_n$  be such that  $S(D_n) = S_n$ . Then,*

$$\text{OPT}(WSC[W, \mathbb{I}, \mathbb{M}, \text{cost}, \text{time}_{oe}[D_n], D_n]) \geq \arg \min_{D \in \mathcal{S}(S_n, n)} \text{cost}(D) \quad (4.10)$$

$$\text{OPT}(WSC[W, \mathbb{I}, \mathbb{M}, \text{cost}, \text{time}_{ue}[D_0], D_0]) \leq \arg \min_{D \in \mathcal{S}(S_n, n)} \text{cost}(D) \quad (4.11)$$

where  $\text{OPT}(\text{prob})$  is optimal cost for a problem "prob".

Weighted set cover problem is a well studied NP-complete problem [25] that admits a linear time greedy algorithm [25] with  $O(\log(n))$  approximation factor, where  $n = |U \setminus U_c|$  is the number of elements to be covered. The key idea is to pick the available set  $a \in A$  that minimizes  $w(a)/e$  where  $e$  is the number of extra elements covered by  $a$ . We use this well studied greedy algorithm to efficiently obtain an approximate solution for the weighted set cover instance defined above. Algorithm-4.1 implements this greedy strategy. In each iteration, the greedy algorithm selects an available set  $a \in A$  that minimizes the cost per extra element covered by it, i.e.  $w(a)/|a \cap T|$  where  $T$  is the target set of elements that are uncovered before that iteration. The algorithm stops when the set of uncovered elements,  $T$ , becomes empty. For indexes  $I \in \mathcal{I}_0$ , this is done by going through each available set in  $A[I]$  (see line-5 algorithm-4.1). For  $I \in \mathcal{I}_0$ ,  $|A[I]| = |M_I|$  where  $M_I \subseteq \mathbb{M}$  is the set of all candidate index mappings of  $I$ . As a result, line-5 of algorithm-4.1 takes  $O(|\mathbb{M}|)$  time. However, there

are  $2^{|M_I|}$  available sets for  $I \in \mathbb{I} \setminus \mathcal{I}_0$ . Therefore, a naive strategy of going through each available set in order to execute an iteration will be too slow for most practical scenarios, especially because the sub-routines  $LB$  and  $UB$  will be called many times during the execution of algorithm-3.2. Therefore, the outer greedy algorithm uses the output of inner greedy algorithm, algorithm-4.2, as an oracle that computes  $\arg \min_{a \in A[I]} w(a)/|a \cap T|$ ,  $I \in \mathbb{I} \setminus \mathcal{I}_0$  efficiently.

---

**Algorithm 4.1** Approximate greedy algorithm for the weighted set cover problem defined in definition-4.2

---

```

1: procedure OUTERGREEDY( $A, cost_{CI}(D_0), U, U_c$ )
2:    $T = U \setminus U_c$  // set of target elements to be covered
3:    $A_g = \{\}$ 
4:   while  $T \neq \phi$  do
5:      $candidates = \{\arg \min_{a \in A[I], I \in \mathcal{I}_0} \frac{w(a)}{|a \cap T|}\}$ 
6:     for  $I \in \mathbb{I} \setminus \mathcal{I}_0$  do
7:        $candidates \cup = \{innerGreedy(A[I], T)\}$ 
8:     end for
9:      $a^* = \arg \min_{a \in candidates} \frac{w(a)}{|a \cap T|}$ 
10:     $T = T \cup a^*$ 
11:     $A_g = A_g \cup \{a^*\}$ 
12:  end while
13:  return  $A_g$ 
14: end procedure

```

---

Algorithm-4.2 breaks the problem of minimizing  $w(a)/|a \cap T|$  into the sub-problems of minimizing  $w(a)/n$ , such that  $|a \cap T| \geq n$ , for all values of  $n \in \{1, 2, \dots, |T|\}$ . Minimizing  $w(a)/n$  is same as minimizing  $w(a)$  for a fixed  $n$ . For  $a \in A[I]$  and  $I \in \mathbb{I} \setminus \mathcal{I}_0$ ,  $w(a)$  can be written as:  $w(a) = maint(I) + \sum_{m \in M_a} time_{CI}[\mathcal{P}_0^m](m)$ . Here  $M_a$  is the set of index mappings of  $I$  used to construct  $a$  and  $\mathcal{P}_0^m$  is the query plan for the query template associated to  $m$  in the initial design  $D_0$ . Now the sub-problem of minimizing  $w(a)/n$  can be written as:

$$\begin{aligned}
M^* = \arg \min_{M \subseteq M_I} & \quad maint(I) + \sum_{m \in M} time_{CI}[\mathcal{P}_0^m](m) \\
s.t. & \quad |(\cup_{m \in M} m(body(m.I))) \cap T| \geq n
\end{aligned} \tag{4.12}$$

Here  $M_I$  is the set of all candidate index mappings of  $I$ . This problem is very similar to the weighted set cover problem where, for each  $m$ , the available sets are  $m(body(m.I))$  with their weights being  $time_{CI}[\mathcal{P}_0^m](m)$ . The only difference between this problem and weighted set cover problem is that the final cover  $M^*$  is only required to cover at least  $n$  elements

of the target set  $T$  instead of covering it completely. This problem is known as "partial" set cover problem [30]. [30] proves that a similar greedy strategy of picking an available set with best per unit cost, in each iteration, gives a log-factor approximation. The body of the "for" loop, in algorithm-4.2, approximates the partial set cover instance shown in equation-4.12 above. It incrementally builds the set  $a$  by adding the index MCDs  $m$  with lowest cost per unit extra elements covered. Line-6 is the key step of the algorithm where it picks index MCD  $m^*$  that minimizes cost per extra element covered by it. The only different between the strategy of algorithm-4.1 and algorithm-4.2 is that we only consider at most  $n - |a \cap T|$  of the extra elements covered by the index MCD  $m$ , i.e. covering more than necessary elements doesn't yield a better per unit cost. [30] proves that this modified greedy algorithm produces a  $O(\log(|T|))$  factor approximate solution for the partial set cover problem. As a result, algorithm-4.2 outputs a  $\log(|T|)$  factor approximate solution for the following problem:  $\min_{a \in A[I]} w(a)/|a \cap T|$ .

---

**Algorithm 4.2** Approximate greedy algorithm for the key step of algorithm-4.1. It outputs an  $\log(|T|)$  factor approximate solution for the problem  $\arg \min_{a \in A[I]} w(a)/|a \cap T|$ ,  $I \in \mathbb{I} \setminus \mathcal{I}_0$ .

---

```

1: procedure INNERGREEDY( $A[I], T$ )
2:    $A_c = \phi$ ,  $w = \{\}$  // candidate sets and their weights
3:   for  $n \in \{1, 2, \dots, |T|\}$  do
4:      $cost = maint(I)$ ,  $a = \phi$ 
5:     while  $|a \cap T| < n$  do
6:        $m^* = \arg \min_{m \in M_I} \frac{time_{CI}[P_0^m](m)}{\min(|(E_{\alpha(CI)}(m) \setminus a) \cap T|, n - |a \cap T|)}$ 
7:        $a = a \cup E_{\alpha(CI)}(m^*)$ 
8:        $cost = cost + time_{CI}[P_0^{m^*}](m^*)$ 
9:     end while
10:     $w(a) = cost$ ,  $A_c = A_c \cup \{a\}$ 
11:  end for
12:   $a^* = \operatorname{argmin}_{a \in A_c} \frac{w(a)}{|a \cap T|}$ 
13:  return  $a^*, w(a^*)$ 
14: end procedure

```

---

[31] proves that the greedy algorithm of the weight set cover problem, with an approximate oracle with approximation factor  $\beta$ , gives an  $O(\beta * \log(n))$  factor approximate solution. Using [31], theorem-4.2 proves the approximation factor of  $\log^2(|W|)$  for the algorithm-4.1 where  $|W|$  is the number of query elements (sub-goals and head variables) in  $W$ . The running time of algorithm-4.2, for any given  $I$ , is  $O(|W|^2 * |M_I|)$  as line-9 takes  $O(|M_I|)$  time and is executed at most  $|T|^2 \leq |W|^2$  times. As a result, the for loop from line-6 to 8 of algorithm-4.1 takes  $O(|W|^2 * |\mathbb{M}|)$  times. Since the body of while loop at line-4 is executed at most

$|W|$  times, the total time complexity of algorithm-4.1 is  $O(|W|^3|\mathbb{M}|)$ .

**Theorem 4.2** *For any set cover instance  $WSC[W, (\mathbb{I}, \mathbb{M}), cost, time_{CI}[D_0], D_0]$ , the cost of the set  $A_g$  selected by algorithm-4.1 is at most  $\log^2(|W|)$  times the optimal cost,*

$$\sum_{s \in A_{greedy}} w(s) \leq \log^2(|W|) * OPT(WSC[W, (\mathbb{I}, \mathbb{M}), cost, time_{CI}[D_0], D_0]) \quad (4.13)$$

Approximate lower and upper bounds for  $UB$  and  $LB$  can now be obtained by running algorithm-4.1. For lower bounds, we need to further divide the cost of output of algorithm-4.1 by  $\log^2(|W|)$  because of theorem-4.2.

## CHAPTER 5: RELATED WORK

The problem of materialized view selection is the most closely related problem to our setting. Materialized view selection problem is to identify views that can precomputed and stored to reduce the query execution time at the expense of some extra space. Most of the approaches for materialized view selection fall into three categories [32]: (i) rewriting based approaches, (ii) AND-OR graphs based approaches, (iii) ad-hoc syntactical analysis of the query workload. [33][34] propose a query rewriting based approach for the class of conjunctive queries (or equivalently select-project-join queries) where the space of materialized views is represented as rewritings of original queries in terms of the candidate view set. Although these approaches are similar to our setting, there are a few fundamental differences. First we consider parameterized queries which behave as efficient function look-ups. As a result, we need to consider the input-output ordering of column of the view in addition to the view definition. Secondly, the system we are design only needs to support a fixed predefined query workload. This is contract to materialized view selection problem where a materialized view is a non-essential augmentation to the system. A database system must be able to operate even in the absence of materialized view. In contrast, index tables are first order citizens in our setting and we cannot not rely on fall back sources for query execution.

The idea of combining the search capabilities on unstructured textual data with structured queries in databases is referred to as DB-IR integration [19][20][21]. Main focus for these works is to embed the traditional inverted index based search system seamlessly into the general framework of relation databases. This can be achieved in a variety ways with their pros and cons. For instance [19] defines a new datatype "text", in addition to standard SQL datatypes, that can store textual information indexed using inverted index. Each tuple acts as a document which is matched with a keyword query. [21] takes a completely different approach where they match keywords in the query with tuples containing them in a column of type string *across* all tables in the database. This may mean joining tables on the fly to construct the full result as the matching tuples may be scattered between different tables. Although our work is tangentially related to this area of research, there major difference in our focus. We focus on designing a search system using databases principles which allows us to design application specific index design using the core concepts of both IR and DB. This may indirectly help in integration of the two technologies but this is not our main focus.

Semi-structured search on combined data[23] refers to retrieval of entities from tagged textual corpus and knowledge bases using keywords based queries [10][11][13]. There different

ways in which entity annotations are used to define queries. [10] Supports queries of type "uw(amazon customer service #phone)" that searches for entities of type "phone" appear alongside the keywords ["amazon", "customer", "service"] within an unordered window of fixed size say 10 words. [13] Uses similar but slightly different query language where they also combine RDF knowledge bases to define a query. For example:- the query "SELECT ?x WHERE type(?x)=plant AND ?x occurs with 'edible leaves' AND ?x native-to Europe" asks for an entity of type plant that occurs with keywords edible leaves and whose attribute "native-to" in the knowledge base has value "Europe". These papers have varied query languages and are more specialized than the ones we are focusing on in this paper. Our query language allows for generalized conjunctive queries with same expressive power as SPJ queries in relational algebra. Our framework allows for generating specialized index designs based on the particularities of the application using the abstraction of query workload. Class of queries considered in this paper is general enough to represent the above queries.

## CHAPTER 6: EXPERIMENTS

We explore two things via experiments. First, we discuss how practical IR applications can be modelled using the proposed framework. For this, we consider three different types of systems: entity search, fixed workload on IMDB like schema, and RDF data. Our second consideration is to explore different possible design choices we could make while implementing the algorithms described above and how they affect the performance. The code used to conduct the experiments can found at: <https://github.com/vmohit/thesis>

In our experiments, we consider the different schema based on binary relations. Table-6.1 lists the details of the two schema which we run experiments on. For example, consider the entity search schema from the table-6.1. It contains 4 different data elements including keywords, entities, documents and entity categories. Within these data elements, there are 3 different base relations. A record  $[k_1, d_1]$  in the base relation keyword-document means that the keyword  $k_1$  is present in the document  $d_1$ . For the purpose of experiments, we populate these base relations randomly. This is done to avoid the effect of particularities of the dataset, which are not considered while building the implementation, to affect the performance of optimization algorithms. Cardinalities shown in the table-6.1 refer to a scale factor of 1. In order to make the dataset bigger or smaller, we multiply each cardinality value with a given scale factor. The cardinalities of each base relation are chosen such that they make sense. For instance, the cardinality of base relation "Movie-Director" is 1500. Now given that there are 1000 different movies and 100 distinct directors, each movie will have around 1.5 directors. Which is to say that some movies will have a single director while other may have 2 (or rarely more than 2 directors). The intuition behind using this schema is to support a fixed query workload efficiently. For instance, say the website of IMDB allows searching for director who have worked in the same movie as a actor. Now since the interface of the website is fixed, the system doesn't have to support arbitrary queries. It just has to support the queries that are linked to the interface of the system.

Using these binary base relations, we construct query templates that take the form of rooted trees. Input to the query templates are the leaves of the rooted trees while the output are the internal nodes. For instance, consider the left most tree in figure-6.1. It represents the entity search queries which can be written as a query template as follows:

$$Q_e[k_1, k_2, c](d, e) : -KD(k_1, d), KD(k_2, d), DE(d, e), CE(c, e) \quad (6.1)$$

Note that each sub-goal in the above query corresponds to an edge in the tree representation shown in figure-6.1. Similarly, each variable in the above query template corresponds to a

Table 6.1: Schema used for conducting experiments. Cardinalities for every data element and base relation is shown for scale factor = 1

Name	Data Elements (cardinality)	Base Relations (cardinality)
Entity Search	Keywords ( $n_k = 1000$ ), Documents ( $n_d = 1000$ ), Entity ( $n_e = 800$ ), Category ( $n_c = 10$ )	Keyword-Document, KD ( $n_{kd} = 50000$ ), Entity-Document, ED ( $n_{ed} = 10000$ ), Entity-Category, EC ( $n_{ec} = 800$ )
IMDB	Actors ( $n_a = 1000$ ), Directors ( $n_d = 100$ ), Movie ( $n_m = 1000$ ), Genre ( $n_g = 10$ ), TV series ( $n_{tv} = 800$ ), Keywords ( $n_k = 1000$ )	Actor-Movie, AM ( $n_{am} = 20000$ ), Movie-Director, MD ( $n_{md} = 1500$ ), Director-TVseries, DT ( $n_{dt} = 1200$ ), Genre-Movie, GM ( $n_{gm} = 2700$ ), Genre-TVseries, GT ( $n_{gt} = 2000$ ), Keyword-Movie, KM ( $n_{km} = 50000$ ), Keyword-TVseries, KT ( $n_{kt} = 40000$ )

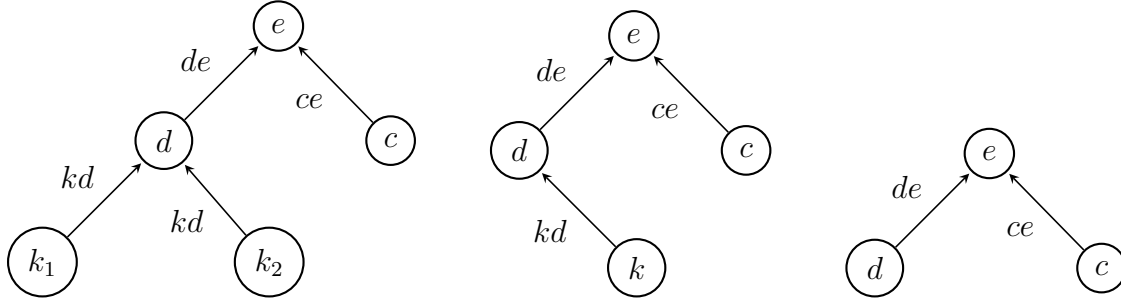


Figure 6.1: From left to right: (i) entity search query template:  $Q_e[k_1, k_2, c](d, e) : -KD(k_1, d), KD(k_2, d), DE(d, e), CE(c, e)$ , (ii) entity-inverted index:  $EINV[k, c](d, e) : -KD(k, d), CE(c, e), DE(d, e)$ , and (iii) document inverted index:  $DINV[d, c](e) : -CE(c, e), DE(d, e)$

node in the tree. Further note that the leaf nodes  $k_1, k_2$  and  $c$  form the input or bound variables of the query while the internal nodes form the target or free variables. We select tree queries for our experiments because they are easier to analyse and lends themselves naturally to the spirit of IR systems. An IR system usually considers queries that retrieve target data items that are related to a few input values. Directed rooted trees provide a general direction of querying from leaf to root node. To test our system thoroughly, we generate random query of different depth and number of nodes. The trees are generated randomly from a bottom up fashion such that the cardinality of each internal node is less than a predefined threshold. This is done to make sure that the execution cost of a query do not increase unboundedly.

Now, given a workload of rooted tree queries, we generate the candidate indexes and index mapping by generating all possible subtrees of depth  $\leq k$ . For our experiments we



select  $k = 3$  where the depth of a leaf node is counted as 1. For the entity search query, we can see that the entity and document inverted index described in [11] form sub-trees of the entity search query. For the index, again the leaf nodes are considered as inputs while the internal nodes are considered as outputs. For instance, the rightmost tree in figure-6.1 describes the documents inverted index  $DINV[d, c](e) : -CE(c, e), DE(d, e)$  that takes input as documents and entity category to obtain the entities of that category present in the input document.

Since directed trees provide a natural flow of query execution from leaf to root node, we don't optimize for the query plan. Instead we assume that a query plan is executed in a bottom up fashion starting from the nodes that are farthest from the root. For instance, following is a possible query plan for the entity search query  $Q_e$ :

$$[INV(k_1, d), INV(k_2, d), DINV(d, c, e)] \quad (6.2)$$

where  $INV[k](d) : -KD(k, d)$  is the inverted index. We select the index mapping  $INV(k_1, d)$  before  $DINV(d, c, e)$  because it covers the edge  $k_1 \rightarrow d$  which is farthest from the root. Because of this, the sequence  $L$  used to describe the branching rule in section-3.2 orders the index mappings with the decreasing order of node to root distances for the nodes covered by it. For instance, the sequence  $L$  for the candidate index mappings  $\mathbb{M} = \{INV(k_1, d), INV(k_2, d), DINV(d, c, e), EINV(k_1, c, e, d), EINV(k_2, c, e, d)\}$  is given as:

$$INV(k_1, d), INV(k_2, d), EINV(k_1, c, e), EINV(k_2, c, e), DINV(d, c, e) \quad (6.3)$$

Because we only consider the plans that sort the index mappings based on their node to root distance, the branch function defined using  $L$  generates all possible index designs. This also allows us to make much better estimates for the context independent cost. For instance, figure-6.2 shows the two context we use to obtain upper and lower bound on the cost contribution of the index mapping  $DINV(d, c, e)$ . The tree on the left describes the subtree that we use to define the over estimated context independent cost of index mapping  $DINV(d, c, e)$ . Consider the state of intermediate result just before we merge the index mapping  $DINV(d, c, e)$  to it in the line-4 of algorithm-2.1. Since the index mappings are generated in the increasing order of node to root distance, the intermediate result must at least subsume the subtree with nodes  $k_1, k_2$  and  $d$  described on the left in figure-6.2. Similarly, to define the lower bounding context independent cost, we consider the biggest subtree of the query tree which corresponds to the intermediate result at the time index mapping  $DINV(d, c, e)$  is merged into it. For that, we consider the full query as the state

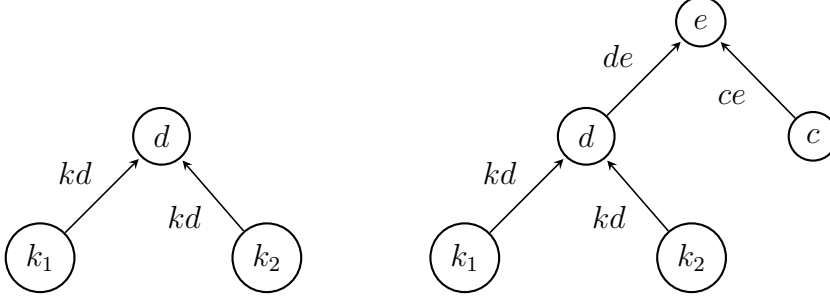


Figure 6.2: Contexts used to estimate  $time_{CI}(DINV(d, c, e))$ . On the left hand side, we show the state of the intermediate result that yields the over estimated cost.

of intermediate result, shown by the tree on right hand side in figure-6.2. Having setup the stage for the experiments, we now move on to the analysis.

**Time complexity of complete state space search:** figure-6.3 shows the effect of various input parameters on the time taken to perform the optimization. We measure the time taken by an algorithm by counting the number of calls made to branch sub-routine in algorithm-3.2 to expand a sub-space into a partition of it.

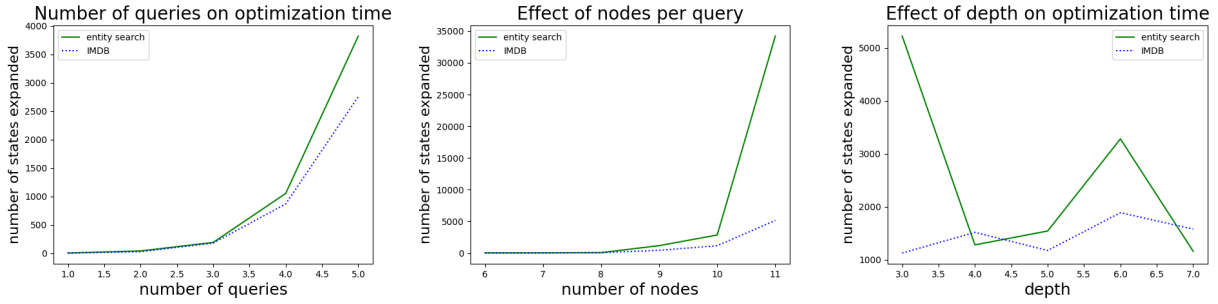


Figure 6.3: Effect of input parameters on the optimization time: the optimization time for an algorithm is measured by the number of times *expand* is called from within algorithm-3.2. The first graph from the left generates  $N \in \{1, 2, 3, 4, 5\}$  random queries depth 3 and number of nodes 4. Note that depth=1 for a leaf node. It then measures the average number of calls made to expand for both the schemas for 3 random trial for each  $N$ . The figure in center shows the increase in number of states expanded w.r.t number of nodes in the query. Number of nodes are varied from 6 to 11. Finally the figure on right measure the number of expansion while increasing the depth of the query from 3 to 8. For all 3 random trials, the query workload contains a single query with 10 nodes.

The number of expansions grow exponentially as the number of queries increase in the workload. This makes sense as growing the number of queries also increase number of index mappings and hence the space of index designs. Next, the increase in number of nodes in a query results in a large increment in number of expansions performed. The pattern however breaks as the increase in the depth of input query, keeping the number of nodes constant,

r

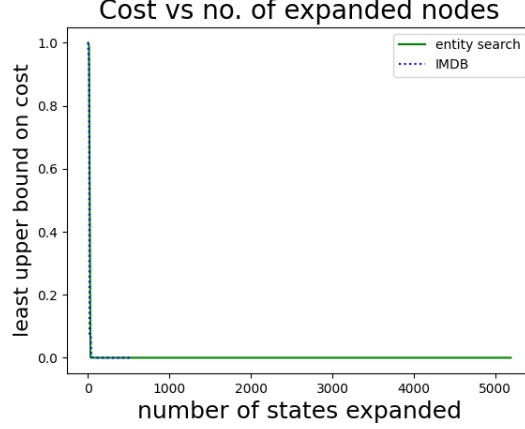


Figure 6.4: Above figure shows the cost of the best known solution during the optimization process,  $Lub$  in algorithm-3.2, w.r.t the number of states which the algorithm has already applied *branch* sub-routine on. To generate this plot, two random queries of depth 3 and number of nodes 7 were used.

results in reduction in the time complexity. This happens because there are fewer possible subtrees, and hence index mappings, in a path as compared to a star. A path with  $n$  nodes only has  $O(n^2)$  distinct connected sub-graphs. However, a start graph with  $n$  nodes has  $2^n$  distinct sub-graphs. Since the time complexity increases exponentially, the exhaustive state-space search algorithm quickly becomes impractical for large query workload. Therefore, we need to improve upon the algorithm-3.2.

In order to better understand the source of exponential time complexity, we take a deeper look at the execution of the algorithm. The figure-6.4 shows the optimal cost ( $Lub$  from algorithm-3.2) during the optimization process at a time by which a given number of calls to *branch* were performed by the algorithm-3.2. Note that the algorithm finds the optimal solution extremely quickly for both the generated random query workload of both the schemas. However, it spends a long time to certify that the best known solution to the algorithm is indeed a global optimum. This takes a lot of time as the algorithm must check for all index designs to be sure that the current solution is optimal.

**Greedy branch expansion:** one reason for the exponential blow-up in the number of expanded designs is that the *branch* subroutine of algorithm-3.2 expands a given sub-space in every possible way. The size of the set *neighbor*, in algorithm-3.1, may reach around  $|\mathbb{M}|$  for many sub-space. Since  $|\mathbb{M}|$  is usually large, this cause an exponential blow up in the number of states which our algorithm must check. Most of these states lead to sub-optimal

solutions. One way to mitigate this issues is to build a small set *neighbor* in a greedy fashion. In our implementation, out of all the neighboring sub-spaces generated by algorithm-3.1, we only consider the ones that have lowest cost to number of goals covered ratio. We sort the *neighbors* set based on cost per number of goals covered and pick only the top  $k$  sub-spaces from the set *neighbors* in algorithm-3.1. We call this hyper parameter branching factor. Figure-6.5 shows the results for this experiment. Notice that the low value of the branching factor, i.e.  $k = 1$ , results in a tiny fraction of number of expanded states as compared to the exhaustive algorithm. Even then it manages to obtain a solution with 1.3 times the cost of the optimal solution. As we increase the value of  $k$ , the cost of the solution obtain decreases but the time required to generate the solution increases.

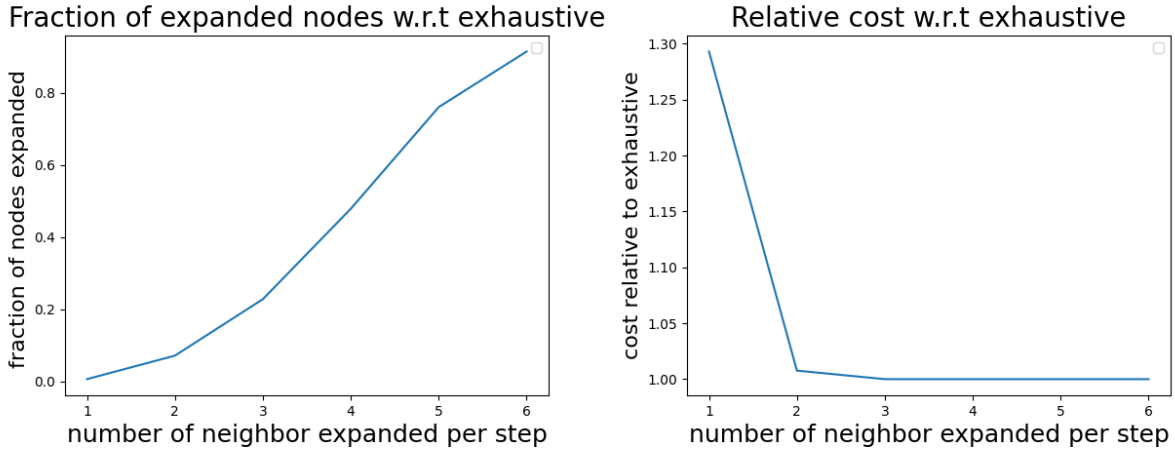


Figure 6.5: In both the plots, x-axis denotes the size of the set *neighbor* constructed during a greedy call to *branch* sub-routine in algorithm-3.1. For instance, if the number of neighbors expanded per step is  $k$ , then select the top- $k$  sub-spaces generated by the algorithm-3.1 and only add those promising sub-spaces to the set *neighbors*. To generate these plots, optimization was perform on 2 random queries of depth 3 and number of nodes 6. The query workload belonged to the IMDB schema from table-6.1

**Pick function:** another factor which may play a crucial role in the performance of the optimization algorithm is the choice of pick function. We experiment with three choices of pick functions: (i) picking the sub-space with least lower bound on the cost, (ii) sub-space with least upper bound, and (iii) subspace that is most complete. To compare the "completeness" of two sub-spaces we compare the number of sub-goals covered by the query plans of the partial index design  $D_n$  associated tot he sub-space  $\mathcal{S}(S(D_n), n)$ . Figure-6.7 shows the performance of optimization algorithm when using different pick functions. Minimizing lower and upper bound costs obtained by the weighted set cover instance defined in chapter-4 produces a extremely good results. In both the case, it is able to find a near optimal solution

very quickly. On the other hand, the pick function that picks the index design with least number of uncovered sub-goals takes some extra time to identify the solution. One possible reason for why the lower and upper bound cost are so good is the restricted class of query templates we experiment with. Since we know that the plans will be executed in a bottom up fashion, we have a good guess to the actual cost contribution of an index mapping. On the other hand, the other pick function relies on obtaining a few good complete index designs early on in the optimal process to be able to prune out the remaining sub-space. However there is no guarantee that the algorithm will stumble across a good index design early on in the process.

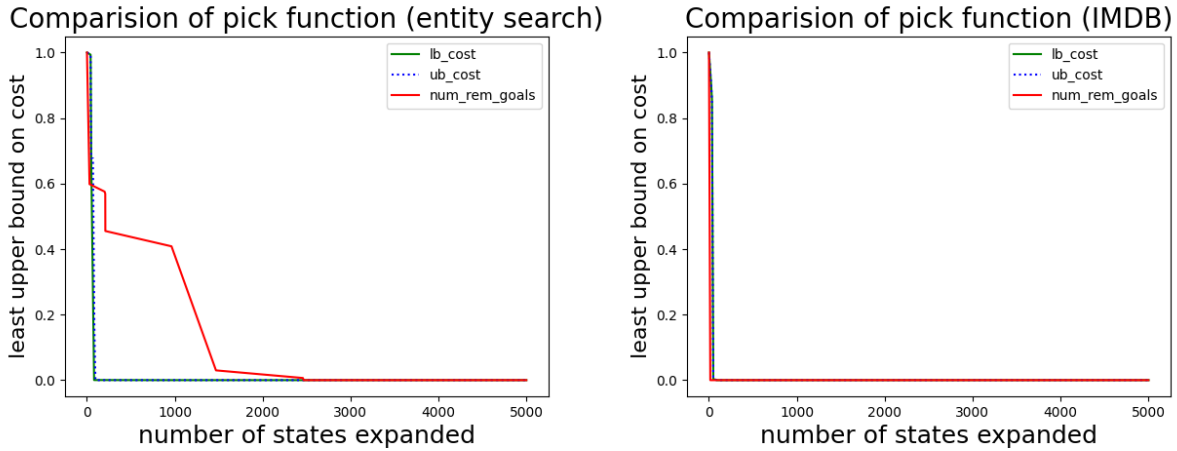


Figure 6.6: The two graphs plot the best known solution at a given time point during the execution of the optimization algorithm for three different *pick* function from algorithm-3.2. The left plot corresponds to the entity search schema and the right one corresponds to the IMDB schema. Both the plots were generated for a workload with 3 queries each having number of nodes 8 and depth 3.

**Reducing weighted set cover calls:** The final strategy we experiment with is to reduce the number of calls to the weight set cover subroutine. Even though the greedy algorithm-4.1 is linear in  $|\mathbb{M}|$ , it may not be optimal to run it frequently because of the  $O(|W|^3)$  factor in its time complexity. In this experiment, we try to reduce the number of times we run the algorithm-4.1 for the weight set cover based relaxation. In order to do that, we observe that the upper and lower bounds for a sub-space  $\mathcal{S}(S_n, n)$  are also valid upper and lower bounds for any sub-space  $\mathcal{S}(S_{n'}, n')$  contained in it. Therefore, we could recycle the lower and upper bound estimates computed for  $\mathcal{S}(S_n, n)$  and just use them for  $\mathcal{S}(S_{n'}, n')$  too. This gives an extremely fast way to obtain the bounds but they will tend to get stale if we don't recompute them. During generation of a index design  $D$ , in *branch* sub-routine of algorithm-3.1, we copy the lower and upper bounds from the parent design  $D$  was generated

from. However, we maintain a count storing the number of times we have copied the lower and upper bounds. In case the this count goes above a threshold say  $l$ , we recompute its lower and upper bounds by running an instance of algorithm-4.1 on it. Increasing this hyper-parameter  $l$ , results in fewer call to weighted set cover algorithm which saves time. However, it results in poorer estimates of upper and lower bound cost which may indirectly result in more exploration of state-space. Because of these two opposing factors, there is expected to be a sweet spot at which the time complexity is minimum.

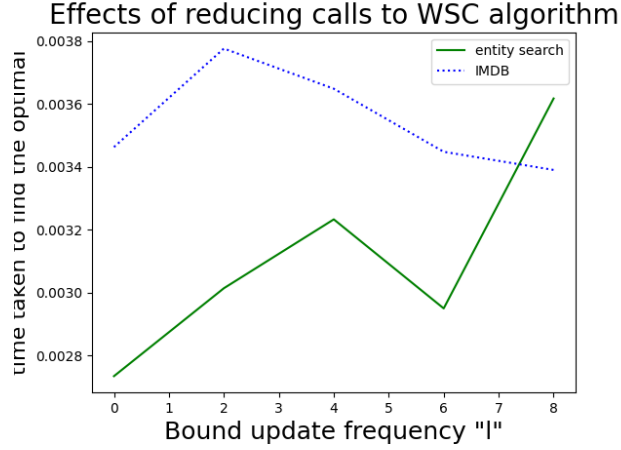


Figure 6.7: The plot compares the time required to obtain the optimal solution for different values of hyper-parameter "l".

The above figure shows the result of optimization process on a workload with 3 queries with each having number nodes 6 and depth 3. In order to compare the time complexity of two approaches, unlike earlier we don't use the number of expanded index designs as a metric. This is because the larger values of "l" are guaranteed to perform more calls to *branch* as the access to high quality bounds has been removed. The above plot compares the performance of the algorithm at different values of "l". For the IMDB schema, it seems to reduce the cost on increment. However there is no reduction in cost over the entity search schema. Even for the IMDB schema, the reduction is not stable and may very well be noise

## CHAPTER 7: CONCLUSION AND FUTURE WORK

In this work, we designed a framework for building IR applications using database concepts. We showed how many of the existing index designs for IR applications can be design within our framework. We formalized the notion of an IR application and its index design. And then designed a concrete algorithm to obtain efficient index designs for IR applications. A common framework to reason about both IR and DB applications opens up many avenues of research. One possible direction of future work would be to extend this framework for more expressive class of queries. Another possibility is to integrate the ideas from IR like top-K retrieval, ranking and scoring which we have not considered in this work.

## REFERENCES

- [1] C. D. Manning, P. Raghavan, and H. Schütze, “Introduction to information retrieval,” 2008.
- [2] E. A. Brewer., “Combining systems and databases: A search engine retrospective.” in *MIT Press*, 2005.
- [3] B. McBride, “The resource description framework (rdf) and its vocabulary description language rdfs,” in *Handbook on Ontologies*, 2004.
- [4] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives, “Dbpedia: A nucleus for a web of open data,” in *Proceedings of the 6th International The Semantic Web and 2nd Asian Conference on Asian Semantic Web Conference*, ser. ISWC’07/ASWC’07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 722–735.
- [5] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, “Freebase: A collaboratively created graph database for structuring human knowledge,” in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’08. New York, NY, USA: Association for Computing Machinery, 2008. [Online]. Available: <https://doi.org/10.1145/1376616.1376746> p. 1247–1250.
- [6] J. Hoffart, F. M. Suchanek, K. Berberich, E. Lewis-Kelham, G. de Melo, and G. Weikum, “Yago2: Exploring and querying world knowledge in time, space, context, and many languages,” in *Proceedings of the 20th International Conference Companion on World Wide Web*, ser. WWW ’11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: <https://doi.org/10.1145/1963192.1963296> p. 229–232.
- [7] “Evgeniy gabrilovich, michael ringgaard, and amarnag subramanya. facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0),” <http://lemurproject.org/clueweb12/>.
- [8] C.-H. Wei, A. Allot, R. Leaman, and Z. Lu, “PubTator central: automated concept annotation for biomedical full text articles,” *Nucleic Acids Research*, vol. 47, no. W1, pp. W587–W593, 05 2019. [Online]. Available: <https://doi.org/10.1093/nar/gkz389>
- [9] “Web data commons (2012),,” <http://webdatacommons.org/>.
- [10] T. Cheng, X. Yan, and K. Chang, “Entityrank: Searching entities directly and holistically,” in *33rd International Conference on Very Large Data Bases, VLDB 2007 - Conference Proceedings*, ser. 33rd International Conference on Very Large Data Bases, VLDB 2007 - Conference Proceedings, J. Gehrke, C. Koch, M. Garofalakis, K. Aberer, C.-C. Kanne, E. Neuhold, V. Ganti, W. Klas, C.-Y. Chan, D. Srivastava, D. Florescu, and A. Deshpande, Eds. Association for Computing Machinery, Inc, 1 2007, pp. 387–398.



- [11] T. Cheng and K. C.-C. Chang, “Beyond pages: supporting efficient, scalable entity search with dual-inversion index,” in *EDBT*, 2010.
- [12] “Balog, k., p. serdyukov, and a. p. de vries (2011). overview of the trec 2011 entity track. in: Trec,” <https://trec.nist.gov/pubs/trec20/papers/ENTITY.OVERVIEW.pdf>.
- [13] H. Bast, F. Baurle, B. Buchhold, and E. Haussmann, “Broccoli: Semantic full-text search at your fingertips,” 07 2012.
- [14] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber, “Ester: efficient search on text, entities, and relations,” *Clarke, Charlie; Fuhr, Norbert; Kando, Noriko; Kraaij, Wessel; de Vries, Arjen P.: SIGIR’07 : 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 671-678 (2007)*, 07 2007.
- [15] H. Bast and B. Buchhold, “Qlever : A ery engine for e icient sparql + text search,” 2017.
- [16] “Sparql query language for rdf,” <https://www.w3.org/TR/rdf-sparql-query/>.
- [17] C. Weiss, P. Karras, and A. Bernstein, “Hexastore: Sextuple indexing for semantic web data management,” *Proc. VLDB Endow.*, vol. 1, no. 1, p. 1008–1019, Aug. 2008. [Online]. Available: <https://doi.org/10.14778/1453856.1453965>
- [18] V. Tablan, K. Bontcheva, I. Roberts, and H. Cunningham, “Mimir: An open-source semantic search framework for interactive information seeking and discovery,” *Journal of Web Semantics*, vol. 30, pp. 52–68, 2015.
- [19] K.-Y. Whang, J.-G. Lee, M.-J. Lee, W.-S. Han, M.-S. Kim, and J.-S. Kim, “Db-ir integration using tight-coupling in the odysseus dbms,” *World Wide Web*, vol. 18, pp. 491–520, 2013.
- [20] S. Chaudhuri, R. Ramakrishnan, and G. Weikum, “Integrating db and ir technologies: What is the sound of one hand clapping?” in *CIDR*, 2005.
- [21] S. Agrawal, S. Chaudhuri, and G. Das, “Dbxplorer: A system for keyword-based search over relational databases,” in *ICDE*, 2002.
- [22] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Greenwich, CT, USA: Manning Publications Co., 2010.
- [23] H. Bast, B. Björn, and E. Haussmann, “Semantic search on text and knowledge bases,” *Found. Trends Inf. Retr.*, vol. 10, no. 2-3, pp. 119–271, June 2016. [Online]. Available: <https://doi.org/10.1561/15000000032>
- [24] E. Balas and P. Toth, “Branch and bound methods for the traveling salesman problem,” Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1983.
- [25] V. V. Vazirani, *Approximation Algorithms*. Berlin, Heidelberg: Springer-Verlag, 2001.

- [26] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.
- [27] S. Deep and P. Koutris, “Compressed representations of conjunctive query results,” in *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. SIGMOD/PODS '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: <https://doi.org/10.1145/3196959.3196979> p. 307–322.
- [28] N. Roussopoulos, “The logical access path schema of a database,” *IEEE Transactions on Software Engineering*, vol. SE-8, no. 6, pp. 563–573, Nov 1982.
- [29] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Pearson Education, 2003.
- [30] P. Slavík, “Improved performance of the greedy algorithm for the minimum set cover and minimum partial cover problems,” 1995.
- [31] P. R. Goundan and A. S. Schulz, “Revisiting the greedy approach to submodular set function maximization,” 2007.
- [32] I. Mami and Z. Bellahsene, “A survey of view selection methods,” *SIGMOD Rec.*, vol. 41, no. 1, pp. 20–29, Apr. 2012. [Online]. Available: <http://doi.acm.org/10.1145/2206869.2206874>
- [33] D. Theodoratos, S. Ligoudistianos, and T. Sellis, “View selection for designing the global data warehouse,” *Data Knowledge Engineering*, vol. 39, no. 3, pp. 219 – 240, 2001, data warehousing. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0169023X01000416>
- [34] F. Goasdoue, K. Karanasos, J. Leblay, and I. Manolescu, “View selection in semantic web databases,” *PVLDB*, vol. 5, pp. 97–108, October 2011. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/view-selection-in-semantic-web-databases/>