Optimizing Index for Taxonomy Keyword Search

Bolin Ding [*]
University of Illinois
bding3@uiuc.edu

Haixun Wang
Microsoft Research Asia
haixunw@microsoft.com

Ruoming Jin [*]
Kent State University
jin@cs.kent.edu

Jiawei Han
University of Illinois
hanj@cs.uiuc.edu

Zhongyuan Wang
Microsoft Research Asia
zhy.wang@microsoft.com

ABSTRACT

Query substitution is an important problem in information retrieval. Much work focuses on how to find substitutes for any given query. In this paper, we study how to efficiently process a keyword query whose substitutes are defined by a given taxonomy. This problem is challenging because each term in a query can have a large number of substitutes, and the original query can be rewritten into any of their combinations. We propose to build an additional index (besides inverted index) to efficiently process queries. For a query workload, we formulate an optimization problem which chooses the additional index structure, aiming at minimizing the query evaluation cost, under given index space constraints. We show the NP-hardness of the problem, and propose a pseudo-polynomial time algorithm using dynamic programming, as well as an $\frac{1}{4}(1-1/e)$-approximation algorithm to solve the problem. Experimental results show that, with only 10% additional index space, our approach can greatly reduce the query evaluation cost.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*

General Terms

Algorithms, Management, Performance

Keywords

Index, taxonomy, keyword search, materialization

[*]Part of the work was done when the authors visited Microsoft Research Asia. The work of Bolin Ding and Jiawei Han was supported in part by the U.S. Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053 (NS-CTA), and NSF IIS-1017362. The work of Ruoming Jin was supported in part by NSF IIS-0953950.

1. INTRODUCTION

Much work has been devoted to query rewriting or finding query substitutions, that is, generating new queries to replace a user's original query [9, 16, 1, 17, 19]. This is motivated by such a simple example: Suppose a user wants to find IT companies in Seattle, and he issues query "IT company, Seattle". However, instead of "IT company", most documents in the collection contain terms like "Microsoft ... Seattle", "Amazon ... Seattle", and "Google ... Seattle". Clearly, unless the system can automatically substitute the term "IT company" in the original query by the terms "Microsoft", "Amazon", etc, at query time, not many informative documents will be retrieved (even in google.com).

For a given word, we can use WordNet [4] to find its synonyms or find terms that have concept-instance relationships with the word. These are good candidates for query substitutes. But WordNet does not contain information such as Microsoft is an IT company, or Kindle is a popular e-Reader. Still, terms such as IT company, Kindle, Microsoft are frequently used in queries. Some recent work (*e.g.*, [26, 21]) focuses on automatically discovering relationships among terms by mining web pages and search engine click logs. This enables us to find important concept-instance relationship for large sets of terms. Table 1 shows a sample of this kind of relationships generated by Probase [26].

In this paper, we assume that we already have complete information of term substitutions, and we will focus on *how to efficiently process a keyword query by answering all of its possible substitutes, and how to optimize the index structure for this purpose.* For a given query, it is likely that there are many eligible substitutes. For example, for the query "IT company acquisition", a document that talks about Microsoft buying Skype is apparently a good match, but this means we need to consider hundreds of IT companies. As another example, for the query "pet, disease", possible substitutes of "pet" may include "cat", "puppy", "dogie", etc, while substitutes of "disease" could be "blastomycosis", "coccidia", "colitis", etc. A crossproduct of the two sets generates a large set of eligible substitutes for the query.

As we mentioned, the *concept-instance* relationship is an important source of query substitutes. The concept-instance relationship defines a *taxonomy.* The *substitution* relationship is a transitive closure of the concept-instance relationship. For example, if "puppy" is an instance of "dog", and "dog" is an instance of "pet", then "puppy" could be a substitute of "pet". If we treat each term as a node, and create for each (concept, instance) pair, an edge from the concept to the instance, then we can think of the taxonomy as a directed tree or forest. For any node that representing a term, its substitute could be any descendant of it in the tree. Figure 1 gives an example of a concept-instance taxonomy.

Contributions. We study indexing methods to efficiently process *taxonomy keyword queries*, or *taxonomy queries* for short. A taxonomy query contains a set of terms. If for any query term, a document either contains the term itself, or contains any of its substitutes, then the document is in the answer to the query. For example, a document containing "cat" and "blastomycosis" is an answer to "pet, disease", and so is a document containing "pet" and "coccidia".

We assume that each term is associated with an *inverted list* (the list of documents containing this term) in the inverted index. A naive way of answering a taxonomy query is to rewrite it as a set of boolean queries (combinations of substitutes for all terms), answer each of them by intersecting the inverted lists of terms in the query, and union the results. This method is inefficient if terms have many substitutes. In the taxonomy query "pet, disease", for example, if "pet" has 100 substitutes and "disease" has 100 substitutes, there are 10000 substitute queries for "pet, disease".

A more efficient method is as follows. For each query term in a taxonomy keyword query, we first compute the union of the inverted lists of all of its substitutes, so as to get a list of documents each of which contains at least one substitute of this term. We call this list of documents the *result list* of the term. Then, we compute the intersection of the result lists of all the query terms as the answer to the query.

There are two questions we need to answer. First, is it worthwhile to *materialize* the result lists (i.e., *precompute and store them as an index*)? Second, which terms should we choose to materialize their result lists? To answer the first question, we extracted and studied two months worth of query log from *Bing.com*, and we find that a large majority (over 85%) of queries contain terms that have a large number of substitutes, for example, "pet" and "IT company". In other words, humans use "high-level" terms frequently in their communication, and thus we can definitely benefit from materializing the result list for these terms. But, we *cannot* materialize result lists of all terms, because the space requirement for index is an important issue. The server farm that maintains the index of a search engine contains thousands of machines. As we can see from the experimental results, materializing result lists for all terms requires 150%-200% *additional* space, and thus 2.5x-3x of number of machines, which is not economically feasible.

The second question – which result lists we should choose to materialize – is the key problem to be addressed in this paper. Many factors are involved to decide the solution. For example, result lists of popular terms are preferred over the ones of rare terms, as popular terms are more frequently queried. Furthermore, the structure of taxonomy offers us more potential to get benefit from materializing result lists of "mid-level" terms, as they can be utilized by more queries.

We propose and solve a *workload-aware index optimization* problem. Given a space budget and a query workload (a query log), we choose and precompute a subset of result lists, so that the space consumption is no more than the budget, and the processing cost of this workload is minimized. This problem is shown to be NP-hard. We propose a pseudo-polynomial time algorithm using dynamic programming, and an $\frac{1}{4}(1 - 1/e)$-approximation algorithm, which works well in practice. More importantly, in experiments on real datasets, we show that, with the materialized result lists carefully selected by our index optimization algorithm, at the cost of only an additional of 10% space, the processing cost can be dramatically reduced, almost to the minimum possible processing cost (when all result lists are materialized, with an additional of 150%-200% space). The proposed algorithms are compared with the simple heuristic which materializes result lists of highly frequent terms in the workload to show their effectiveness.

Organization. Section 2 introduces the concept of taxonomy and formally defines taxonomy keyword queries and their answers. In Section 3, we first introduce how we process taxonomy queries and the associated cost models, and then formalize our workload-aware index optimization problem. Section 4 presents our approaches to solve this index optimization problem. Experimental results are reported in Section 5, followed by discussion and extension of our techniques in Section 6, and related work in Section 7.

2. TAXONOMY KEYWORD SEARCH

Unlike previous work that focuses on how to find substitutes for an original query [9, 16, 1, 17, 19], we focus on how to efficiently process a *taxonomy keyword query* whose substitutes are defined based on a given *taxonomy*. In this section, we formally define *taxonomy* and *taxonomy keyword queries*, and discuss challenges of processing taxonomy keyword queries

2.1 Taxonomy

A taxonomy $(\mathcal{T}, \sqsubseteq)$ consists of a universe of terms $\mathcal{T}$ and a term-term *concept-instance relationship* $\sqsubseteq$.

Concept-instance relationship. The *concept-instance relationship* $\sqsubseteq$ is a partial order $\mathcal{T}$. For two terms t_1 and t_2, we write $t_1 \sqsubseteq t_2$ or $t_2 \sqsupseteq t_1$, if t_1 is an *instance* of t_2 (or t_2 is a *concept* of t_1). For example, "JPEG" is an instance of "file format", and thus we write $t_1 =$ "JPEG" $\sqsubseteq$ "file format" $= t_2$. Table 1 lists some terms in our taxonomy and their instances.

The taxonomy we use is known as Probase [26]. It contains concept-instance relationships obtained from a web corpus of 1.68 billion documents using syntactic patterns including Hearst patterns [7]. From a sentence "...fruits *such as* apple, banana, and blueberry ...," *e.g.*, we know that "apple", "banana", and "blueberry" are instances of "fruit", where *such as* is one of the Hearst patterns. When a term appears in a user's query, we assume she/he may also be interested in the instances of this term. So, instances are possible substitutes for a query term. For details of how the taxonomy is constructed, we refer the readers to [26, 13].

Substitution relationship. The *substitution relationship* $\preceq$ is defined to be the *transitive closure* of the relationship $\sqsubseteq$. For two terms t_1 and t_2, t_1 is an *substitute* of t_2, denoted as $t_1 \preceq t_2$, iff $t_1 = t_2$, or $t_1 \sqsubseteq t_2$, or there exists $t_{i_1}, \ldots, t_{i_k}$ s.t. $t_1 \sqsubseteq t_{i_1} \sqsubseteq \ldots \sqsubseteq t_{i_k} \sqsubseteq t_2$. For example, "puppy" $\sqsubseteq$ "dog" $\sqsubseteq$ "pet" implies "puppy" $\preceq$ "pet". When a user types "pet", we assume that she/he may be also interested in "puppy".

Later in this paper, we also use $\mathcal{S}(t)$ to denote all the substitutes of the term t: $\mathcal{S}(t) = \{t' \in \mathcal{T} \mid t' \preceq t\}$.

Taxonomy tree and substitution relationship. Given a taxonomy $(\mathcal{T}, \sqsubseteq)$, a directed graph is constructed by creating a node for each term $t \in \mathcal{T}$, and a directed edge $t \to t'$ iff

Concept	Typical Instances
actor	Tom Hanks, Marlon Brando, ...
airline	British Airways, Deltae, ...
chemical compound	carbon dioxide, phenanthrene, ...
disease	AIDS, Alzheimer, chlamydia, ...
company	IBM, Microsoft, Google, ...
file format	PDF, JPEG, TIFF, ...

Table 1: Samples of concept-term relationship

$t' \sqsubseteq t$. Abusing the notation a bit, we use $(\mathcal{T}, \sqsubseteq)$ to denote both the taxonomy and the resulting directed graph.

We will focus on the case that $(\mathcal{T}, \sqsubseteq)$ is a *directed tree or forest* in the major part of this paper, and discuss how our techniques can be extended for general taxonomies in Section 6.

Obviously, we have $t_1 \preceq t_2$ iff t_1 is a descendant of t_2 in the taxonomy tree $(\mathcal{T}, \sqsubseteq)$, and $\mathcal{S}(t)$ is essentially the set of descendants of t, including t itself, in the tree.

EXAMPLE 2.1. *Figure 1 shows an example of taxonomy, represented as a tree, where concept-instance relationship $\sqsubseteq$ is represented as edges. In this taxonomy, term t_2 has three instances: $t_3, t_6, t_9 \sqsubseteq t_2$. The substitutes of a term are its descendants, together with itself, in this tree. Term t_2 has seven instances: $t_2, t_3, \ldots, t_9 \preceq t_2$, i.e., $\mathcal{S}(t_2) = \{t_2, t_3, \ldots, t_9\}$.*

2.2 Taxonomy Keyword Queries

Inverted lists. Let $\mathcal{D}$ be a collection of documents, where each document consists of terms from a vocabulary $\mathcal{T}$. We assume that an *inverted index* is constructed for document retrieval. For any term $t \in \mathcal{T}$, the *inverted list* $\mathcal{I}(t) \subseteq \mathcal{D}$ is the set of all documents containing the term t.

Result lists. The *result list* $\mathcal{R}(t)$ of a term t denotes the set of documents containing any substitute of t.:

$$\mathcal{R}(t) = \bigcup_{s \in \mathcal{S}(t)} \mathcal{I}(s). \tag{1}$$

The intuition behind $\mathcal{R}(t)$ is, when a user refers to t, we assume she may also be interested in all documents containing any of its substitutes $\mathcal{S}(t)$. For example, suppose "Windows7" $\sqsubseteq$ "MS Windows" $\sqsubseteq$ "operating system", when "operating system" appears in the query, the user is also interested in the documents containing "MS Windows" or "Windows7".

Answers to a taxonomy keyword query Now we formally define the answer to a *taxonomy keyword query* q as follows. Given a taxonomy query in the form of $q = \{t_1, t_2, \ldots, t_k\}$, we can find the answers to q as follows:

$$\mathcal{R}(q) = \bigcap_{i=1}^{k} \mathcal{R}(t_i) = \bigcap_{i=1}^{k} \left(\bigcup_{s \in \mathcal{S}(t_i)} \mathcal{I}(s) \right) \tag{2}$$

$$= \bigcup_{\{s_1,\ldots,s_m\} \in \mathcal{S}(t_1) \times \ldots \times \mathcal{S}(t_k)} \left(\bigcap_{i=1}^{k} \mathcal{I}(s_i) \right). \tag{3}$$

Our goal is to evaluate $\mathcal{R}(q)$ as efficiently as possible.

2.3 Challenges

We introduce two naive approaches and their scalability issues. The first approach retrieves $\mathcal{I}(s)$ for each instance s of each term t_i, and evaluate the disjunctive normal form as in (2). As an example, consider the taxonomy shown in Figure 1. Given a query $q = \{t_2, t_{10}, t_{17}\}$, according to Equation (2), we need to evaluate $\mathcal{R}(q) = \mathcal{R}(t_2) \cap \mathcal{R}(t_{10}) \cap \mathcal{R}(t_{17})$, which means we need to evaluate $(\mathcal{I}(t_2) \cup \mathcal{I}(t_3) \cup \ldots \cup \mathcal{I}(t_9)) \cap (\mathcal{I}(t_{10}) \cup \mathcal{I}(t_{11}) \cup \ldots \cup \mathcal{I}(t_{15})) \cap (\mathcal{I}(t_{17}) \cup \mathcal{I}(t_{18}) \cup \ldots \cup \mathcal{I}(t_{21})$. However, when some t_i has a large number of instances, that is, $|\mathcal{S}(t_i)|$ is large, evaluating the disjunctive normal form becomes very inefficient, as we need to retrieve a large number of inverted lists, and perform many union operations.

In the other naive approach, we precompute and store $\mathcal{R}(t) = \bigcup_{s \in \mathcal{S}(t)} \mathcal{I}(s)$ for all terms t's in the offline stage; and then in the online stage, we evaluate a taxonomy query $q = \{t_1, \ldots, t_k\}$ by intersecting the $\mathcal{R}(t_i)$'s directly. This approach does not need to evaluate unions of $\mathcal{I}(s)$'s during query time, but an apparent drawback is that it requires excessive amount of storage for $\mathcal{R}(t)$'s in a non-trivial taxonomy $(\mathcal{T}, \sqsubseteq)$.

In the rest part of this paper, we introduce an approach that precomputes $\mathcal{R}(t)$'s for a selected set $\mathbf{P}$ of terms t's. We will discuss how to evaluate a taxonomy query q by taking advantage of these precomputed lists, and how to select $\mathbf{P}$.

3. PROCESSING TAXONOMY QUERY

In this section, we first propose a query processing model for taxonomy queries and an associated cost model: we propose to precompute and store the result lists $\mathcal{R}(t)$'s for a subset of terms $\mathbf{P} \subseteq \mathcal{T}$ offline, and discuss how (much) $\mathbf{P}$ can improve the efficiency of online query processing. We then introduce the problem of *workload-aware index optimization*: given a query log $\mathcal{Q}$ and a space budget, how to select the set $\mathbf{P}$ so that the query processing cost is minimized.

3.1 Query Processing and Cost Model

Offline preprocessing. For each term $t \in \mathcal{T}$, we assume $\mathcal{I}(t)$, the inverted list of documents containing t, is precomputed and stored, as in most IR systems. However, it is too space-consuming to also store *result lists* $\mathcal{R}(t)$'s for all terms in $\mathcal{T}$, as the total size of $\mathcal{R}(t)$'s is usually larger than the total size of $\mathcal{I}(t)$'s. We select a subset of terms $\mathbf{P} \subseteq \mathcal{T}$, and precompute and store $\mathcal{R}(t) = \cup_{s \in \mathcal{S}(t)} \mathcal{I}(s)$ for each $t \in \mathbf{P}$. We will define the problem of how to select $\mathbf{P}$ later in Section 3.2.

Once $\mathbf{P}$ is fixed, we can compute $\mathcal{R}(t)$'s for $t \in \mathbf{P}$ by merging lists together, from bottom to top, starting with $\mathcal{I}(t')$'s.

We can use either linear lists or hash tables to implement $\mathcal{I}(t)$ and $\mathcal{R}(t)$, although both are called "lists". This determines how the query processing algorithm accesses these sets.

In general, for a term set $\mathbf{P} \subseteq \mathcal{T}$, the total space we need to materialize lists $\mathcal{R}(t)$'s for all terms $t \in \mathbf{P}$ is proportional to

$$\mathsf{space}(\mathbf{P}) = \sum_{t \in \mathbf{P}} |\mathcal{R}(t)|. \tag{4}$$

Online query processing. For an online taxonomy keyword query $q = \{t_1, \ldots, t_k\}$, we want to take advantage of the precomputed lists $\mathcal{R}(t)$'s (for $t \in \mathbf{P}$) as much as pos-

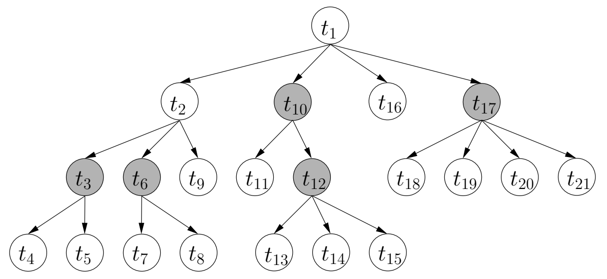


Figure 1: A taxonomy and $\mathbf{P} = \{t_3, t_6, t_{10}, t_{12}, t_{17}\}$

sible. Recall the query answer $\mathcal{R}(q)$ can be computed as $\cap_{i=1}^{k}\mathcal{R}(t_i)$.

```
PROCESSQUERY($q$, $\mathbf{P}$)
 1: For each $t_i \in q$ do:
 2:       $\mathcal{L}(t_i) \leftarrow$ PROCESSTERM($t_i, \mathbf{P}$).
 In linear scan model:
 3: Compute $\mathcal{R}(t_i)$ from the set of lists $\mathcal{L}(t_i)$ $(i = 1, \ldots, k)$.
 4: Return $\mathcal{R}(q) = \cap_{i=1}^{k}\mathcal{R}(t_i)$.
 Or, in hash lookup model: (initially, $\mathcal{R}(q) = \varnothing$)
 5: Compute $\mathcal{R}(t_1)$ from the set of lists $\mathcal{L}(t_1)$.
 6: For each $d \in \mathcal{R}(t_1)$ do
 7:       For each $\mathcal{L}(t_i)$ $(i = 2, \ldots, k)$ check:
          whether $d \in L$ for some list $L \in \mathcal{L}(t_i)$;
 8:       If "yes" for all $i = 2, \ldots, k$, $\mathcal{R}(q) \leftarrow \mathcal{R}(q) \cup \{d\}$.
 9: Return $\mathcal{R}(q)$.
PROCESSTERM($t$, $\mathbf{P}$)
(It returns pointers to the lists $\mathcal{I}(t')$'s and $\mathcal{R}(t')$'s as $\mathcal{L}$)
 1: If $t \in \mathbf{P}$ then return $\mathcal{L}(t) \leftarrow \{\mathcal{R}(t)\}$;
 2: Else: $\mathcal{L} = \{\mathcal{I}(t)\}$;
 3:       For each child $t'$ of $t$: $\mathcal{L} \leftarrow \mathcal{L} \cup$ PROCESSTERM($t'$);
 4: Return $\mathcal{L}(t) \leftarrow \mathcal{L}$.
```

Algorithm 1: PROCESSQUERY and PROCESSTERM

EXAMPLE 3.1. *Consider the taxonomy in Figure 1 and* $\mathbf{P} = \{t_3, t_6, t_{10}, t_{12}, t_{17}\}$ *(gray nodes in the figure). Assume we want to evaluate the taxonomy query* $q = \{t_2, t_{10}\}$. *To compute* $\mathcal{R}(t_2)$, *according its definition (1),* $\mathcal{R}(t_2) = \cup_{j=2}^{9}\mathcal{I}(t_j)$. *Since* t_3 *and* t_6 *are in* $\mathbf{P}$, *we can get* $\mathcal{R}(t_3)$ *and* $\mathcal{R}(t_6)$ *directly, and ignore their substitutes* t_4, t_5, t_7, *and* t_8. *Thus,* $\mathcal{R}(t_2) = \mathcal{R}(t_3) \cup \mathcal{R}(t_6) \cup \mathcal{I}(t_2) \cup \mathcal{I}(t_9)$. *To compute* $\mathcal{R}(t_{10})$, *since* t_{10} *is in* $\mathbf{P}$, *we can get* $\mathcal{R}(t_{10})$ *directly, and ignore its substitutes* t_{11} *and* t_{12}. *So* $q = \{t_2, t_{10}\}$ *can be evaluated as:* $(\mathcal{R}(t_3) \cup \mathcal{R}(t_6) \cup \mathcal{I}(t_2) \cup \mathcal{I}(t_9)) \cap (\mathcal{R}(t_{10}))$.

From the above example, to evaluate $\mathcal{R}(t_i)$, if $t_i \in \mathbf{P}$, we can directly retrieve $\mathcal{R}(t_i)$ from the index; otherwise, we retrieve $\mathcal{I}(t_i)$, and *recursively* check t_i's child t'–the union of lists needed to compute $\mathcal{R}(t')$ for all children t''s together with $\mathcal{I}(t_i)$ is exactly $\mathcal{R}(t_i)$. This is how PROCESSTERM(t) in Algorithm 1 works. After PROCESSTERM(t_i) finishes, $\mathcal{L}_i$ is the set of pointers to lists (either $\mathcal{I}(t')$ or $\mathcal{R}(t')$ for a term) that are needed to compute $\mathcal{R}(t_i)$. In Example 3.1, we have $\mathcal{L}(t_2) = \{\mathcal{R}(t_3), \mathcal{R}(t_6), \mathcal{I}(t_2), \mathcal{I}(t_9)\}$ and $\mathcal{L}(t_{10}) = \{\mathcal{R}(t_{10})\}$.

Later in this section, we will formally describe $\mathcal{L}(t)$, which is returned by PROCESSTERM, and how to compute $\mathcal{R}(t_i)$ using lists in $\mathcal{L}(t_i)$'s in both *linear scan* model and *hash lookup* model. For this purpose, we introduce $C(t, \mathbf{P})$ and $\overline{C}(t, \mathbf{P})$:

Intuitively, any term t' in $C(t, \mathbf{P})$ is the closest descendant of t on the path from t to t' in $\mathbf{P}$. In other words, for any $t' \in C(t, \mathbf{P})$, there is no any other term in $\mathbf{P}$ as both t's descendant and t''s ancestor. When PROCESSTERM touches any term in $C(t, \mathbf{P})$, it stops recursion and includes $\mathcal{R}(t)$ in $\mathcal{L}$. In Example 3.1, $C(t_{10}, \mathbf{P}) = \{t_{10}\}$ and $C(t_2, \mathbf{P}) = \{t_3, t_6\}$.

Furthermore, $\overline{C}(t, \mathbf{P})$ records those terms that are substitutes (descendants) of t (in $\mathcal{S}(t)$) but not covered by $C(t, \mathbf{P})$. When PROCESSTERM accesses those terms in $\overline{C}(t, \mathbf{P})$, it adds $\mathcal{I}(t)$ to $\mathcal{L}$ and continue the traversal. In Example 3.1, we have $\overline{C}(t_2, \mathbf{P}) = \{t_2, t_9\}$ and $\overline{C}(t_{10}, \mathbf{P}) = \varnothing$. Finally, we note that if t itself in $\mathbf{P}$, then $C(t, \mathbf{P}) = \{t\}$ and $\overline{C}(t, \mathbf{P}) = \emptyset$.

$$C(t, \mathbf{P}) = \{t' \in \mathbf{P} \mid t' \preceq t \wedge \nexists x \in \mathbf{P} : t' \prec x \preceq t\}, \quad (5)$$
$$\overline{C}(t, \mathbf{P}) = \mathcal{S}(t) - \cup_{x \in C(t, \mathbf{P})}\mathcal{S}(x). \quad (6)$$

Given this, we can formally write $\mathcal{L}(t)$ and $\mathcal{R}(t)$ as follows:

$$\mathcal{L}(t) = \{\mathcal{R}(t') \mid t' \in C(t, \mathbf{P}\} \cup \{\mathcal{I}(t'') \mid t'' \in \overline{C}(t, \mathbf{P}\}, \quad (7)$$

$$\mathcal{R}(t) = \left(\cup_{t' \in C(t,P)}\mathcal{R}(t')\right) \bigcup \left(\cup_{t'' \in \overline{C}(t,P)}\mathcal{I}(t'')\right). \quad (8)$$

PROPOSITION 1. *With the selection of* $C(t, \mathbf{P})$ *in Equation (5) and* $\overline{C}(t, \mathbf{P})$ *in Equation (6),* $\mathcal{R}(t)$ *can be computed as in (8) with the minimum number of union operations.*

Depending on how lists $\mathcal{I}(t)$ and $\mathcal{R}(t)$ are implemented, as *linear lists* or *hash tables*, we mainly consider the following two types of query processing algorithms, and quantify the cost to process one query q for a selection of term set $\mathbf{P}$.

The linear scan model (lines 3-4 in PROCESSQUERY). We first consider the model where elements in $\mathcal{I}(t)$ and (materialized) $\mathcal{R}(t)$ are stored as *sorted linear lists*, which support sequential access of elements in $\mathcal{I}(t)$ and $\mathcal{R}(t)$ in certain order, and might be compressed. For a query $q = \{t_1, t_2, \ldots, t_k\}$, we evaluate $\mathcal{R}(q)$ in two steps:

(i) We first compute $\mathcal{R}(t_i)$ for each $t_i \in q$, according to (8): we scan the lists $\mathcal{R}(t')$'s and $\mathcal{I}(t'')$'s in $\mathcal{L}(t_i)$, and merge them into the sorted list $\mathcal{R}(t_i)$, with duplicates eliminated. (ii) We then take the intersection of the sorted lists $\mathcal{R}(t_i)$'s $(i = 1, 2, \ldots, k)$, using the linear scan, as the answer $\mathcal{R}(q)$.

Since the total number of operations in (i) is proportional to the number of elements we accessed in $\mathcal{R}(t')$'s and $\mathcal{I}(t'')$, the cost of computing $\mathcal{R}(t_i)$ can be quantified as the total size of lists involved in Equation (8), i.e.,

$$\mathsf{cost}_s(t, \mathbf{P}) = \sum_{t' \in C(t, \mathbf{P})} |\mathcal{R}(t')| + \sum_{t'' \in \overline{C}(t, \mathbf{P})} |\mathcal{I}(t'')|. \quad (9)$$

The cost in (ii) is independent on the selection of $\mathbf{P}$, so we quantify the total cost of processing q for a fixed $\mathbf{P}$ as:

$$\mathsf{cost}_s(q, \mathbf{P}) = \sum_{t_i \in q} \mathsf{cost}_s(t_i, \mathbf{P}). \quad (10)$$

$\mathsf{cost}_s(q, \mathbf{P})$ is essentially the total number of elements we need to access in the index structure to answer the query q. It can be noted that there could be lots of variants of the above algorithm with different practical performance. However, we claim that Equation (10) captures the dominating factor of the cost, because if only sequential accesses are allowed for each list $\mathcal{I}(t)$ or $\mathcal{R}(t)$, to compute the result $\mathcal{R}(q)$, we need to scan each element in $\mathcal{I}(t)$ or $\mathcal{R}(t)$ at least once.

As Equation (10) is the total size of these lists we need to access, it can be used to approximate the actual processing cost for answering q.

The hash lookup model (lines 5-9 in ProcessQuery). We then consider that model where each list $\mathcal{I}(t)$ or $\mathcal{R}(t)$ is preprocessed in such a way that hash lookup (search in $O(1)$ time) is supported. In this model, we compute $\mathcal{R}(q)$ as follows.

Given a query $q = \{t_1, t_2, \ldots, t_k\}$, we find the term t_0 in q whose instances (including itself) appear in the minimum number of documents, i.e., $t_0 = \arg\min_{t_i \in q} |\mathcal{R}(t_i)|$. W.l.o.g., suppose $t_0 = t_1$. We first compute $\mathcal{R}(t_1)$, by adding each element in each $\mathcal{R}(t')$ or $\mathcal{I}(t'') \in \mathcal{L}(t_1)$ into the hash table $\mathcal{R}(t_1)$ to eliminate duplicates. Then, for each element (document ID) d in $\mathcal{R}(t_1)$, we use hash lookups to check whether it also appears in every other $\mathcal{R}(t_i)$'s–if yes, we include it into the query answer $\mathcal{R}(q)$. In particular, to check whether d is in $\mathcal{R}(t_i)$, according to (8), we need to check whether it is in some $\mathcal{R}(t')$ (for some $t' \in C(t_i, \mathbf{P})$) or in some $\mathcal{I}(t'')$ (for some $s' \in \overline{C}(t_i, \mathbf{P})$). The number of hash lookups we need to check whether each d is in $\mathcal{R}(t)$ is at most

$$\mathcal{N}(t, \mathbf{P}) = |C(t, \mathbf{P})| + |\overline{C}(t, \mathbf{P})|. \tag{11}$$

And thus for all elements in $\mathcal{R}(t_1)$, we need at most

$$\mathsf{cost}_h(q, P) = |\mathcal{R}(t_1)| \cdot \sum_{t_i \in q} \mathcal{N}(t_i, \mathbf{P}) \tag{12}$$

hash lookups in total. Since the query processing cost in this model is dominated by the number hash lookups, we now use $\mathsf{cost}_h(q, \mathbf{P})$ to quantify the processing cost.

It can be noted that, on average, we may not need as many hash lookups as in Equation (12), because once we find a candidate does not exist in $\mathcal{R}(t_j)$, we do not need to verify its existence in other lists any more. However, Equation (12) is usually proportional to the actual number of hash lookups needed, and thus characterizes the cost of query processing.

Binary search and other query processing models. There is another type of query processing algorithms based on binary search. As shown in the experiments in [3], they perform similar to the ones based on hash lookup. So we can also quantify their cost using Equation (12). Detailed analysis is omitted.

3.2 Problem of Index Optimization

We introduce the *workload-aware index optimization* problem. The goal is to select $\mathbf{P}$ to minimize the query processing cost for a workload of queries $\mathcal{Q}$, subject to space budget.

Given a space budget B_0, we precompute and store $\mathcal{R}(t)$'s for a subset of terms $\mathbf{P} \subseteq \mathcal{T}$, s.t. they consume no more than B_0 space, and the cost of processing queries in $\mathcal{Q}$ can be minimized. More specifically, we are interested in the *expected query processing cost* of $\mathcal{Q}$, i.e., for a randomly-picked query from $\mathcal{Q}$, the expected cost to process it. This cost can be computed as the (weighted) average of costs for all queries in $\mathcal{Q}$:

$$\mathsf{cost}_{\exp}(\mathcal{Q}, \mathbf{P}) = \sum_{q \in \mathcal{Q}} \mathsf{cost}(q, \mathbf{P}) \cdot w(q), \tag{13}$$

where $\mathsf{cost}(q, \mathbf{P})$ could be either $\mathsf{cost}_s(q, \mathbf{P})$ in (9)-(10) or $\mathsf{cost}_h(q, \mathbf{P})$ in (11)-(12), depending on the query processing model we are using, and $w(q)$ is the frequency of query q.

We note that the cost function (13) may involve the cardinalities $|\mathcal{R}(t)|$'s and $|\mathcal{I}(t)|$'s as in (9). We need to precompute these cardinalities (without actual construction of lists $\mathcal{R}(t)$'s) to obtain the value of the cost function for a selection of $\mathbf{P}$. This is done at the time when the inverted index $\{\mathcal{I}(t)\text{'s}\}$ is constructed: when a term t in a document is scanned, besides adding the document ID into $\mathcal{I}(t)$, for each ancestor t' of t in the taxonomy, increase the counter for $|\mathcal{R}(t')|$ by one.

Problem statement (IndexSelection)
In the IndexSelection *problem, suppose we have a collection of documents $\mathcal{D}$ for which a taxonomy $(\mathcal{T}, \sqsubseteq)$ is predefined. Given a query log $\mathcal{Q}$ and a space budget B_0, our goal is to find a subset of terms $\mathbf{P} \subseteq \mathcal{T}$, and the objective is to*

$$\begin{aligned} \textit{minimize} \quad & \mathsf{cost}_{\exp}(\mathcal{Q}, \mathbf{P}) \\ \textit{s.t.} \quad & \mathsf{space}(\mathbf{P}) \le B_0, \quad \mathbf{P} \subseteq \mathcal{T}. \end{aligned}$$

Theorem 2. *The* IndexSelection *problem is HP-hard, where $|\mathcal{T}|$ is the size of the input to the problem.*

Proof. (Sketch) We reduce Knapsack to IndexSelection. An instance of Knapsack is: given n items with weights $\{w_i\}$ and values $\{v_i\}$, we want to select a subset of items with total weight $\le B_0$, so that the total value is maximized. In the linear scan model, we construct an instance of IndexSelection: There are n queries in $\mathcal{Q} = \{q_1, \ldots, q_n\}$, and each query $q_i = \{t_i\}$ ($t_i \ne t_j$ for $i \ne j$). For term t_i, we have $\mathcal{S}(t_i) = \{t_i, t'_i\}$, and $\mathcal{S}(t'_i) = \{t'_i\}$ ($t'_i \notin \{t_1, \ldots, t_n\}$). Let $|\mathcal{I}(t_i)| = w_i$, $|\mathcal{I}(t'_i)| = v_i$, and $\mathcal{I}(t_i) \supseteq \mathcal{I}(t'_i)$ (so $|\mathcal{R}(t_i)| = w_i$). To complete the proof, we need to prove that, with the same B_0 in two problems, the optimal solution to Knapsack is OPT if and only if the optimal solution to IndexSelection is $\sum_{i=1}^{n}(w_i + v_i) - \text{OPT}$. A similar reduction from Partition can be done for the hash lookup model and we omit the details for the space limit. □

4. INDEX OPTIMIZATION

A naive algorithm

A simple solution for IndexSelection is to include the top-l most frequent terms into $\mathbf{P}$, where l is selected to be the max one so that the space budget B_0 is not violated. Here, the *frequency* of a term t means the total frequency of queries in $\mathcal{Q}$ that contain t. Intuitively, if a term t appears more often in $\mathcal{Q}$, we get more benefit from materializing $\mathcal{R}(t)$, and thus we include it into $\mathbf{P}$ earlier.

This naive algorithm, however, is usually sub-optimal. There are two basic reasons. First, frequency is not the only criteria to quantify the benefit of including a term into $\mathbf{P}$. For example, a term t may have low frequency but it is a substitute of many other frequent terms; in this case, we still can benefit a lot by materializing $\mathcal{R}(t)$. Second, the benefit of materializing $\mathcal{R}(t)$ is not additive but dependent on the terms already in $\mathbf{P}$. In other words, for example, the benefit of including t into $\mathbf{P}$ when $\mathbf{P} = \varnothing$ is different from the benefit when $\mathbf{P} = \{t_1\}$.

Overview of our algorithms

To get the optimal solution for IndexSelection, a strawman approach is to enumerate all subsets of $\mathcal{T}$ as $\mathbf{P}$, which needs $O(2^N)$ time, where $N = |\mathcal{T}|$ is the total number of terms in the taxonomy.

We first introduce how to use dynamic programming to get the optimal solution in $O(N \cdot B_0 \cdot 2^h)$ time, where h is the height of the taxonomy. It is still an exponential-time algorithm (recall the NP-hardness of IndexSelection in Theorem 2). However, since h is usually small in practice, this dynamic programming algorithm works reasonably well.

Interestingly, we find *benefit function*, which evaluates the processing performance gain of materializing $\mathcal{R}(t)$'s for terms in $\mathbf{P}$, has a nice property, "diminishing returns", or *submodularity*: i.e., the benefit of including t into $\mathbf{P}_0$ when $\mathbf{P}_0 = \mathbf{P}_1$ is always no less than the benefit when $\mathbf{P}_0 = \mathbf{P}_2$, if $\mathbf{P}_1 \subseteq \mathbf{P}_2$. Thanks to this property, a fast greedy algorithm with provable *approximation ratio* can be designed, as we show next.

In the rest of this section, we first formally define the benefit function in Section 4.1, and introduce and analyze the two index optimization algorithms, dynamic programming and greedy, in Sections 4.2 and 4.3, respectively.

4.1 Benefit Function

For the ease of discussion and analysis, before introducing the two algorithms, we first derive a *benefit* function gain by rewriting the cost function ($\mathsf{cost}_{\exp}$) in IndexSelection.

The linear scan model. From (9)-(10), we rewrite (13) as:

$$\begin{aligned}\mathsf{cost}_{\exp}(\mathcal{Q}, \mathbf{P}) &= \sum_{q \in \mathcal{Q}} \sum_{t \in q} \mathsf{cost}_s(t, \mathbf{P}) \cdot w(q) \\ &= \sum_{t \in \mathcal{T}} \left(\mathsf{cost}_s(t, \mathbf{P}) \cdot \sum_{q \in \mathcal{Q}: t \in q} w(q) \right) \\ &= \sum_{t \in \mathcal{T}} \mathsf{cost}_s(t, \mathbf{P}) \cdot w_s(t, \mathcal{Q}), \qquad (14)\end{aligned}$$

where $w_s(t, \mathcal{Q}) = \sum_{q \in \mathcal{Q}: t \in q} w(q)$ is the *frequency* of t and $\mathsf{cost}_s(t, \mathbf{P}) = \sum_{t' \in C(t, \mathbf{P})} |\mathcal{R}(t')| + \sum_{t'' \in \overline{C}(t, \mathbf{P})} |\mathcal{I}(t'')|$, as in (9), is the number of elements to access to compute $\mathcal{R}(t)$.

The hash lookup model. For a query $q = \{t_1, \ldots, t_k\}$, let $\mathcal{C}(q) = |\mathcal{R}(t_1)|$ be the number of candidates. From (11)-(12),

$$\begin{aligned}\mathsf{cost}_{\exp}(\mathcal{Q}, \mathbf{P}) &= \sum_{q \in \mathcal{Q}} \sum_{t \in q} \mathcal{C}(q) \cdot \mathcal{N}(t, \mathbf{P}) \cdot w(q) \\ &= \sum_{t \in \mathcal{T}} \left(\mathcal{N}(t, \mathbf{P}) \cdot \sum_{q \in \mathcal{Q}: t \in q} \mathcal{C}(q) \cdot w(q) \right) \\ &= \sum_{t \in \mathcal{T}} \mathcal{N}(t, \mathbf{P}) \cdot w_h(t, \mathcal{Q}), \qquad (15)\end{aligned}$$

where we have $w_h(t, \mathcal{Q}) = \sum_{q \in \mathcal{Q}: t \in q} \mathcal{C}(q) \cdot w(q)$ and $\mathcal{N}(t, \mathbf{P}) = |C(t, \mathbf{P})| + |\overline{C}(t, \mathbf{P})|$ as in (11) is the max number of hash lookups needed to check whether a candidate is in $\mathcal{R}(t)$.

Function gain. The cost functions in Equations (14) and (15) are in the same form. Now we define the *benefit function* $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$ for the purpose of our optimization:

$$\mathsf{gain}(\mathcal{Q}, \mathbf{P}) = \mathsf{cost}_{\exp}(\mathcal{Q}, \varnothing) - \mathsf{cost}_{\exp}(\mathcal{Q}, \mathbf{P}). \qquad (16)$$

Obviously, minimizing $\mathsf{cost}_{\exp}(\mathcal{Q}, \mathbf{P})$ is equivalent to maximizing $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$. According to (14)-(15), we define:

$$\mathsf{gain}(t, \mathbf{P}) = \mathsf{cost}_s(t, \varnothing) - \mathsf{cost}_s(t, \mathbf{P}), \text{ or} \qquad (17)$$
$$\mathsf{gain}(t, \mathbf{P}) = \mathcal{N}(t, \varnothing) - \mathcal{N}(t, \mathbf{P}). \qquad (18)$$

for the two models, and thus, we have

$$\mathsf{gain}(\mathcal{Q}, \mathbf{P}) = \sum_{t \in \mathcal{T}} \mathsf{gain}(t, \mathbf{P}) \cdot w(t, \mathcal{Q}), \text{ where} \qquad (19)$$
$$w(t, \mathcal{Q}) = w_s(t, \mathcal{Q}) \text{ or } w_h(t, \mathcal{Q}), \text{ respectively.}$$

4.2 Dynamic Programming Algorithm

For a taxonomy $(\mathcal{T}, \sqsubseteq)$, we introduce the dynamic programming algorithm to maximize $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$. For a set of terms $\mathbf{P}$ and a term t', let $\mathbf{P} + t'$ denote $\mathbf{P} \cup \{t'\}$. Let $\mathcal{A}(t)$ be the set of ancestors of t in the taxonomy plus t itself.

We first suppose all terms in $\mathcal{T}$ are ordered as the *pre-order traversal* sequence: $\mathcal{T} = \{t_1, t_2, \ldots, t_N\}$ ($N = |\mathcal{T}|$). Refer to Figure 1 for example. Given this, the straightforward dynamic programming considers each term (starting from the last t_N) has two options, including it in $\mathbf{P}$ or excluding it. For the former, it needs to solve the case for $\mathcal{T} = \{t_1, t_2, \ldots t_{N-1}\}$ with budget $B_0 - |\mathcal{R}(t_N)|\}$; and for the latter, it needs to solve the case for $\mathcal{T} = \{t_1, t_2, \ldots t_{N-1}\}$ with budget B_0. Recursively, it can produce the optimal results with time complexity $O(2^N)$. Clearly, this is prohibitively expensive and inefficient as there are too many sub-problems being reused.

Subproblem definition. An observation which forms the basis of dynamic programming is as follows: *For any term $t_i \in \mathbf{P}$, the materialization of $\mathcal{R}(t_i)$ only benefits the nodes on the path going from t_i up to the term whose direct ancestor (in the taxonomy tree) is in* $\mathbf{P}$. For instance, in Figure 1, $\mathcal{R}(t_{12})$ only benefits itself, and $\mathcal{R}(t_3)$ benefits itself and both t_2 and t_1.

However, how to utilize this property to define and reuse subproblems in dynamic programming is not trivial. This is because for a term (*e.g.*, t_{21} in Figure 1), the choice of whether adding it into $\mathbf{P}$ is not only related to its closest ancestor (t_{17} w.r.t. t_{21}) in $\mathbf{P}$; but also, the choice on this term and the ones on its cousins (t_{20}, t_{19}, and t_{18} w.r.t. t_{21}) are correlated as they share a set of common ancestors ($\{t_{17}, t_1\}$ in this example).

We introduce two lemmas below which help define the subproblems in the dynamic programming.

Lemma 3. *Let $\mathcal{A}(t_i)$ be the set of ancestors of t_i in the taxonomy plus t_i. Numbering nodes in $\mathcal{T}$ in the order as the pre-order traversal, a property of this numbering is: for any two consecutive t_{i-1} and t_i, we have $\mathcal{A}(t_{i-1}) \supseteq \mathcal{A}(t_i) - \{t_i\}$.*

In general, for any set $\mathbf{P} \subseteq \mathcal{T}$ of terms, define:

$$\Delta\mathsf{gain}_{\mathbf{P}}(t') = \mathsf{gain}(\mathcal{Q}, \mathbf{P} + t') - \mathsf{gain}(\mathcal{Q}, \mathbf{P}). \qquad (20)$$

Here let us order the terms in a set $\mathbf{P}$ as a subsequence of $\mathcal{T}$, $\mathbf{P} = \{t_{i_1}, t_{i_2}, \ldots, t_{i_m}\}$ ($i_1 < \ldots < i_m$). Let $\mathbf{P}_j = \{t_{i_1}, \ldots, t_{i_j}\}$ be the first j terms in $\mathbf{P}$. We can rewrite $\mathsf{gain}(\mathcal{Q}, \mathbf{P}) =$

$$\Delta\mathsf{gain}_{\varnothing}(t_{i_1}) + \Delta\mathsf{gain}_{\mathbf{P}_1}(t_{i_2}) + \ldots + \Delta\mathsf{gain}_{\mathbf{P}_{m-1}}(t_{i_m}).$$

Recall our observation that the materialization of $\mathcal{R}(t_i)$ only benefits t_i itself up to the term whose direct ancestor in $\mathbf{P}$. So as long as the choices (whether or not to be included in $\mathbf{P}$) on the ancestors are made, the marginal benefit of materializing $\mathcal{R}(t_i)$ is fixed. We can prove this claim formally in Lemma 4. Details are in Appendix A.1.

Lemma 4. *For any $\mathbf{P}', \mathbf{P}'' \subseteq \{t_1, t_2, \ldots, t_{i-1}\} \subseteq \mathcal{T}$, if $\mathbf{P}' \cap \mathcal{A}(t_i) = \mathbf{P}'' \cap \mathcal{A}(t_i)$, then $\Delta\mathsf{gain}_{\mathbf{P}'}(t_i) = \Delta\mathsf{gain}_{\mathbf{P}''}(t_i)$.*

From Lemmas 3-4, *for $\mathbf{P} \subseteq \{t_1, \ldots, t_{i-1}\}$, let $S_{i-1} = \mathbf{P} \cap \mathcal{A}(t_{i-1})$ be the set of t_{i-1}'s ancestors, including t_{i-1} itself, which are included in $\mathbf{P}$, and then the value of $\Delta\mathsf{gain}_{\mathbf{P}}(t_i)$ is determined by S_{i-1}, for any fixed i.* The reason is as follows. From Lemma 3, we know that, if $S_{i-1} = \mathbf{P} \cap \mathcal{A}(t_{i-1})$ is fixed, then $\mathbf{P} \cap \mathcal{A}(t_i)$ is also fixed. So from Lemma 4, $\Delta\mathsf{gain}_{\mathbf{P}}(t_i)$ is uniquely determined by S_{i-1} for fixed i.

Therefore, we can write $\Delta\mathsf{gain}_{\mathbf{P}}(t_i)$ as $\Delta\mathsf{gain}_{S_{i-1}}(t_i)$ with S_{i-1} defined above. We will introduce how to compute the value of $\Delta\mathsf{gain}_{\mathbf{P}}(t_i) = \Delta\mathsf{gain}_{S_{i-1}}(t_i)$ for both the linear scan model and the hash lookup model in Appendix A.1.

Based on the above discussion, for each term t_i and a fixed budget B, we can partition all the possible choices of $\mathbf{P}$ on the terms $\{t_1, \ldots, t_{i-1}\}$, whose number is in the order of $O(2^{i-1})$, into the equivalent classes defined by S_{i-1}, whose number is in the order of $O(2^h)$. Here, h is the height of the taxonomy and is much less than $|\mathcal{T}|$. More formally, the subproblem is defined as: *finding the best solution $P(i) \subseteq \{t_1, t_2, \ldots, t_i\}$ which uses space B with $P(i) \cap \mathcal{A}(t_i) = S_i$.*

Algorithm description. We are ready to introduce our dynamic programming algorithm in this space of subproblems. Let $\mathrm{OPT}_i(B, S_i)$ be the value of the best partial solution $\mathbf{P}(i) \subseteq \{t_1, t_2, \ldots, t_i\}$, among those which spend space B, i.e., $\sum_{t \in \mathbf{P}(i)} |\mathcal{R}(t)| = B$, and with $\mathbf{P}(i) \cap \mathcal{A}(t_i) = S_i$. We can derive the following recursion for $\mathrm{OPT}_i(B, S_i)$:

$$\mathrm{OPT}_i(B, S_i) = \max_{S_{i-1}:\ S_{i-1} \cap \mathcal{A}(t_i) \subseteq S_i} \tag{21}$$

$$\begin{cases} \mathrm{OPT}_{i-1}(B, S_{i-1}) & \text{if } t_i \notin S_i, \\ \mathrm{OPT}_{i-1}(B - |\mathcal{R}(t_i)|, S_{i-1}) + \Delta\mathsf{gain}_{S_{i-1}}(t_i) & \text{if } t_i \in S_i. \end{cases}$$

And the initial condition is: $\mathrm{OPT}_1(B, S_1) =$

$$\begin{cases} 0 & B = 0, S_1 = \varnothing \\ \Delta\mathsf{gain}_{\varnothing}(\{t_1\}) & B = |\mathcal{R}(t_1)|, S_1 = \{t_1\} \\ -\infty & \text{otherwise.} \end{cases} \tag{22}$$

Here, $-\infty$ means that the other settings of (B, S_1) are invalid.

LEMMA 5. *Equations* (21)-(22) *can correctly compute the optimal solution for the* INDEXSECTION *problem.*

PROOF. The initial condition (22) is obviously correct from the definition. The recursion (21) considers two cases: if $t_i \notin S_i$, both the space consumption and the solution value are unchanged; and if $t_i \in S_i$, it needs additional $|\mathcal{R}(t_i)|$ space, and increases the benefit of the solution $\mathbf{P}(i)$ by $\Delta\mathsf{gain}_{\mathbf{P}(i-1)}(t_i) = \Delta\mathsf{gain}_{S_{i-1}}(t_i)$, which depends only on S_{i-1}. (21) essentially enumerates all settings of S_{i-1} to get the best $\mathbf{P}(i)$. □

The dynamic programming algorithm to compute recursions (21)-(22) can be implemented as Algorithm 2 in a backward way: start from $\mathrm{OPT}_1(B, S_1)$, using $\mathrm{OPT}_i(B, S_i)$ to update $\mathrm{OPT}_{i+1}(B', S_{i+1})$. Lines 5-6 consider the case $t_{i+1} \notin S_{i+1}$ as in the first case of (21), and lines 7-8 consider the case $t_{i+1} \in S_{i+1}$ as in the second case of (21).

THEOREM 6. *Algorithm 3 correctly gets the optimal solution to the* INDEXSELECTION *problem in $O(N \cdot B_0 \cdot 2^h \cdot h)$ time and $O(N \cdot B_0 \cdot 2^h)$ space, where N is the total number of terms in $\mathcal{T}$, and h is the height of the taxonomy tree.*

PROOF. The correctness is from Lemma 5 and the above discussion. To store $\mathrm{OPT}_i(B, S_i)$ for all settings of (i, B, S_i),

```
Input: Terms $\mathcal{T}$, query log $\mathcal{Q}$, and space budget $B_0$.
 1: Initialize $\mathrm{OPT}_1(B, S_1)$ according to (22).
 2: Initialize $\mathrm{OPT}_i(B, S_i)$ as $-\infty$ for any $i > 1$ and $B$.
 3: For $i = 1$ to $N - 1$ do
    For $B = 0$ to $B_0$ do
    For each subset $S_i$ in $\mathcal{A}(t_i)$ do
 4:   If $\mathrm{OPT}_i(B, S_i) \neq -\infty$ then
 5:      $S_{i+1} \leftarrow S_i \cap \mathcal{A}(t_{i+1})$,
         $B' \leftarrow B$;
 6:      $\mathrm{OPT}_{i+1}(B', S_{i+1}) \leftarrow \max\{\mathrm{OPT}_{i+1}(B', S_{i+1}),$
                                   $\mathrm{OPT}_i(B, S_i)\}$;
 7:      $S_{i+1} \leftarrow (S_i \cap \mathcal{A}(t_{i+1})) \cup \{t_{i+1}\}$,
         $B' \leftarrow B + |\mathcal{R}(t_{i+1})|$;
 8:      $\mathrm{OPT}_{i+1}(B', S_{i+1}) \leftarrow \max\{\mathrm{OPT}_{i+1}(B', S_{i+1}),$
                                   $\mathrm{OPT}_i(B, S_i) + \Delta\mathsf{gain}_{S_i}(t_{i+1})\}$;
 9: Return $\max\{\mathrm{OPT}_N(B, S_N) \mid B \leq B_0, S_N \subseteq \mathcal{A}(t_N)\}$.
```

Algorithm 2: DYNAMICPROGRAM

we need $O(N \cdot B_0 \cdot 2^h)$ space, as $1 \leq i \leq N$, $0 \leq B \leq B_0$, and S_i could be any subset of a term's ancestors in the taxonomy tree. As for the time complexity, the cost of each iteration of lines 4-8 is dominated by the computation of $\Delta\mathsf{gain}_{S_i}(t_{i+1})$ (discussed in Appendix A.1) and S_{i+1}, both of which need $O(h)$ time. And there are totally $O(N \cdot B_0 \cdot 2^h)$ iterations. □

We note that Algorithm 2 is a pseudo-polynomial time algorithm, even when the tree height h is a constant, because B_0 may not be polynomially-bounded in N. However, for fixed h, if applying the rounding-scaleup idea for the KNAPSACK problem [25], we can modify DYNAMICPROGRAM to yield an FPTAS (fully polynomial-time approximation scheme) for the INDEXSELECTION problem. It gets an $(1 + \epsilon)$-approximation with running time polynomial in N and $1/\epsilon$

Implementation. Based on similar rounding-scaleup ideas, in practice, we can scale down the space budget $B_{\max} \leftarrow \lfloor B_0/\alpha \rfloor$ and consider terms (whether or not to be included into $\mathbf{P}$) with heights no larger than $h_{\max}$ in the taxonomy. $B_{\max}$ and $h_{\max}$ are selected s.t. the $O(N \cdot B_{\max} \cdot 2^{h_{\max}})$ space is affordable. Also, the space consumption of each term t is scaled down as $b(t) \leftarrow \lceil |\mathcal{R}(t)|/\alpha \rceil$, and let $\mathsf{space}(\mathbf{P}) = \sum_{t \in \mathbf{P}} b(t)$. We apply Algorithm 2 on this scaled-down instance, and can get an approximate solution to INDEXSELECTION, while reducing its time complexity to $O(N \cdot B_{\max} \cdot 2^{h_{\max}} \cdot h_{\max})$.

4.3 Submodularity and Greedy Algorithm

The complexity of the dynamic programming algorithm in Section 4.2 could be prohibitive when B_0 is large. Now we introduce a simpler but more efficient algorithm, which provide an approximate solution to the INDEXSELECTION problem. It is based on two properties of the function $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$.

(i) $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$ is *submodular*, i.e., demonstrates "diminishing returns", w.r.t. $\mathbf{P}$. The benefit of including a term t' into $\mathbf{P}$ decreases as $\mathbf{P}$ grows. Formally, fixing $\mathcal{Q}$, iff $\mathbf{P}_1 \supseteq \mathbf{P}_2$,

$$\begin{aligned} &\Delta\mathsf{gain}_{\mathbf{P}_1}(t') = \mathsf{gain}(\mathcal{Q}, \mathbf{P}_1 + t') - \mathsf{gain}(\mathcal{Q}, \mathbf{P}_1) \\ \leq &\Delta\mathsf{gain}_{\mathbf{P}_2}(t') = \mathsf{gain}(\mathcal{Q}, \mathbf{P}_2 + t') - \mathsf{gain}(\mathcal{Q}, \mathbf{P}_2). \end{aligned}$$

(ii) $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$ is *monotone*, i.e., we can always get benefit from including additional terms into $\mathbf{P}$. Formally, fixing $\mathcal{Q}$, $\mathsf{gain}(\mathcal{Q}, \mathbf{P}_1) \geq \mathsf{gain}(\mathcal{Q}, \mathbf{P}_2)$, iff $\mathbf{P}_1 \supseteq \mathbf{P}_2$.

LEMMA 7. *For any fixed t, $\mathsf{gain}(t, \mathbf{P})$ in* (17)-(18) *w.r.t. $\mathbf{P}$ is submodular and monotone for both the linear scan and*

```
Input: Terms $\mathcal{T}$, query log $\mathcal{Q}$, and space budget $B_0$.
1: Initially, let $\mathbf{P} \leftarrow \emptyset$.
2: While $\mathsf{space}(\mathbf{P}) \leq B_0$ do:
3:     $t_0 \leftarrow \arg\max_{t \notin \mathbf{P}} \left(\mathsf{val}(t) = \frac{\mathsf{gain}(\mathcal{Q},\mathbf{P}+t)-\mathsf{gain}(\mathcal{Q},\mathbf{P})}{|\mathcal{R}(t)|}\right)$.
4:     $\mathbf{P} \leftarrow \mathbf{P} \cup \{t_0\}$.
5: Return the better one of $\mathbf{P} - \{t_0\}$ and $\{t_0\}$ as the solution.
```

Algorithm 3: GreedySelect

the hash lookup model. And thus $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$ *in* (19) *w.r.t.* $\mathbf{P}$ *(for any fixed* $\mathcal{Q}$*) is submodular and monotone in both models.*

Proof. The monotonicity is easy to be verified. After precomputing $\mathcal{R}(t')$ and adding t' into $\mathbf{P}$, when the query processing algorithm reaches t', it can retrieve $\mathcal{R}(t')$ directly, instead of going down to the children of t' to retrieve more lists (refer to line 1 of ProcessTerm in Algorithm 1). So from (17)-(18), (9), and (11), fixing t, the value of $\mathsf{gain}(t, \mathbf{P})$ can only increase if we adding any term t' into $\mathbf{P}$.

For the submodularity of gain, refer to Appendix A.2. □

From the submodularity and the monotonicity of the function $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$ (fixing $\mathcal{Q}$), the greedy algorithm GreedySelection (Algorithm 3) works well. We assume that $|\mathcal{R}(t)| \leq B_0$ for every $t \in \mathcal{T}$ (i.e., never include a term t into $\mathbf{P}$ with $|\mathcal{R}(t)| > B_0$). Initially, let $\mathbf{P}$ be empty. In each iteration of lines 2-4, we pick the term t_0 with the *unit value*

$$\mathsf{val}(t) = \frac{\Delta\mathsf{gain}_{\mathbf{P}}(t)}{|\mathcal{R}(t)|} = \frac{\mathsf{gain}(\mathcal{Q}, \mathbf{P} + t) - \mathsf{gain}(\mathcal{Q}, \mathbf{P})}{|\mathcal{R}(t)|}$$

to be the max among all terms that are not in $\mathbf{P}$ yet, and put t_0 into $\mathbf{P}$. We repeat this greedy selection until the space needed for terms in $\mathbf{P}$ reaches the budget B_0, i.e., $\sum_{t \in \mathbf{P}} |\mathcal{R}(t)| > B_0$. Eventually, let t_0 be the last selected term, and we return the best of $\mathbf{P} - \{t_0\}$ and $\{t_0\}$ as our the solution $\mathbf{P}_0$.

Theorem 8. *Suppose* $\mathbf{P}^*$ *is the optimal solution to the* IndexSelection *problem, and* $\mathbf{P}_0$ *is the solution returned by* GreedySelect*, we have* $\mathsf{gain}(\mathcal{Q}, \mathbf{P}_0) \geq \frac{1}{4}(1 - 1/e) \cdot \mathsf{gain}(\mathcal{Q}, \mathbf{P}^*)$*. It needs* $O(N^2 \cdot h)$ *time and* $O(N)$ *space, where* N *is the number of terms, and* h *is the height of the taxonomy.*

Proof. From Lemma 7, $\mathbf{P}$ on line 5 is at least as good as the solution obtained from Algorithm 1 in [11], so from [11], we have $\mathsf{gain}(\mathcal{Q}, \mathbf{P}) \geq \frac{1}{2}(1 - 1/e) \cdot \mathsf{gain}(\mathcal{Q}, \mathbf{P}^*)$. Also from the submodularity, $\mathsf{gain}(\mathcal{Q}, \mathbf{P} - \{t_0\}) + \mathsf{gain}(\mathcal{Q}, \{t_0\}) \geq \mathsf{gain}(\mathcal{Q}, \mathbf{P})$. So we have at least one of $\mathsf{gain}(\mathcal{Q}, \mathbf{P} - \{t_0\})$ and $\mathsf{gain}(\mathcal{Q}, \{t_0\})$ no less than $\frac{1}{4}(1-1/e) \cdot \mathsf{gain}(\mathcal{Q}, \mathbf{P}^*)$. As we assume $\forall t \in \mathcal{T} : |\mathcal{R}(t)| \leq B_0$, $\{t_0\}$ could also be a solution. So the approximation ratio follows. For the space complexity, we need only $O(1)$ space for each term t in GreedySelect. As for the time complexity, the iteration of lines 2-4 repeats at most N times, and in each iteration, we need $O(N \cdot h)$ time to select the best t_0 (we will discuss in Appendix A.1 about how to compute $\Delta\mathsf{gain}_{\mathbf{P}}(t)$ in $O(h)$ time). □

We note that, in practice, GreedySelect is much more efficient than $O(N^2 \cdot h)$, because lines 2-4 usually repeat much less than N times (it stops as soon as $\mathsf{space}(\mathbf{P}) > B_0$). That is also the reason why we do not apply Algorithm 1 in [11], which needs exactly N iterations. In our problem, t_0 has to be selected in at least $\Theta(N \cdot h)$ time and N is large, so early termination on line 2 improves the practical performance a lot.

In fact, from the submodularity of $\mathsf{gain}(t, \cdot)$ and [23], there is an $(1 - 1/e)$-approximation algorithm with running time $\Theta(N^4)$ [23]. Note that an $\Theta(N^4)$-time algorithm is too slow in our context, as N could be very large in practice. So we will only test GreedySelect in the experiments.

5. EXPERIMENTS

We evaluate our index optimization techniques for taxonomy keyword queries using real datasets. There are two baselines: NoInd, which does NOT materialize $\mathcal{R}(t)$ for any term ($\mathbf{P} = \varnothing$), and AllInd, which materializes $\mathcal{R}(t)$'s for all terms ($\mathbf{P} = \mathcal{T}$). In our taxonomy query processing model, these two methods consume the minimal and maximal possible space, respectively. Given a space budget B_0, the naive algorithm introduced at the beginning of Section 4 is denoted as Naive, the DynamicProgram algorithm in Section 4.2 is denoted as DP, and the GreedySelect algorithm in Section 4.3 is denoted as Greedy, in the following experiments.

To have the DynamicProgram algorithm work in practice, we set parameters $B_{\max}$ and $h_{\max}$ (discussed at the end of Section 4.2) as: $B_{\max} = 400$ and $h_{\max} = 7$. We found this is the best setting (minimal processing cost) for our dataset.

Algorithms are coded in C# and evaluated on a 48GB 64-bit 2.67GHz PC. All index structures are stored in the memory.

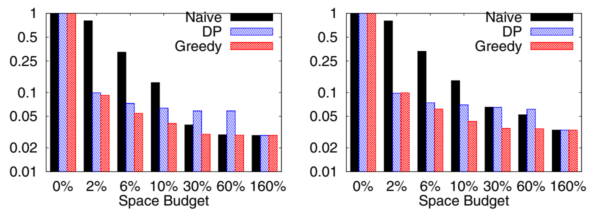
Datasets. Our techniques take i) a taxonomy $(\mathcal{T}, \sqsubseteq)$, ii) a query log $\mathcal{Q}$, and iii) a collection of documents $\mathcal{D}$ as the input, for the purpose of index optimization (i.e., selecting the term set $\mathbf{P}$ for materializing result lists $\mathcal{R}(t)$'s).

i) We use a taxonomy automatically extracted from a corpus of 1.68 billion web pages, using algorithms proposed in [26, 21]. This taxonomy contains 279,109 terms, with term-term concept-instance relationship predefined.

ii) The query log is extracted from queries to search engine *Bing.com* from Sept 2007 to Feb 2010, denoted as $\mathcal{Q}_0$, and we only keep the ones with frequency larger than 300. We have 1,260,526 different queries which appear totally more than 6 billion times. To test how our index optimization techniques work for future (unseen) queries, i.e., the ones not in the query log, we also extract 6 query workloads, denoted as $\mathcal{Q}_1, \mathcal{Q}_2, \ldots, \mathcal{Q}_6$, from Mar, Apr, May, Jun, Jul, and Aug, respectively, later in 2010.

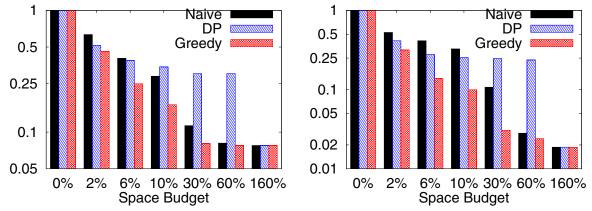
iii) With a sample from *Bing.com*'s web page corpus as the collection of documents, we build the inverted index, i.e., $\mathcal{I}(\cdot)$, on these pages for all terms in our taxonomy. The inverted lists $\mathcal{I}(t)$'s contain a total of 10,282,150 entries (40MB memory). To test the scalability of our index optimization techniques, we also extract a larger sample whose inverted index has a total of 250M entries (1GB memory). If we compute and materialize all the result lists $\mathcal{R}(t)$'s, we need 150%-200% memory *in additional to* the inverted index (250%-300% in total).

Measures. We record the *time* and the *number of (linear scan or hash lookup) operations* needed to process workloads of queries. When we compare different index optimization techniques, we usually normalize the time and the number of operations to the ones in NoInd (i.e., $\mathbf{P} = \varnothing$ does not



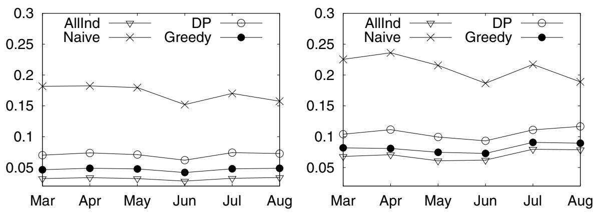
(a) # Operations (ratio) (b) Processing time (ratio)

Figure 2: Linear scan model: varying space budget



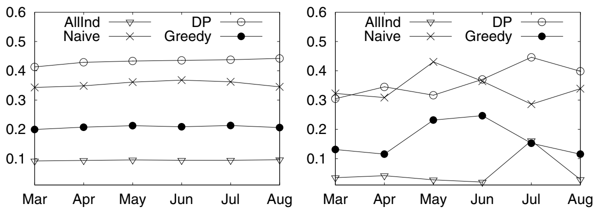
(a) # Operations (ratio) (b) Processing time (ratio)

Figure 3: Hash lookup model: varying space budget



(a) # Operations (ratio) (b) Processing time (ratio)

Figure 4: Linear scan model: future queries



(a) # Operations (ratio) (b) Processing time (ratio)

Figure 5: Hash lookup model: future queries

materialize any $\mathcal{R}(t)$), so in this case, a ratio in (0,1] is reported.

Exp-I: Varying space budget B_0. Our first experiment is to test the proposed index optimization techniques for different space budgets. We use the 40MB inverted index and vary the space budget B_0 from 2% $\times$ 40MB to 160% $\times$ 40MB. We use $\mathcal{Q}_0$ for both index optimization and reporting query processing time/# operations. Materializing all $\mathcal{R}(t)$'s needs 160% additional memory in this case. The result is reported in Figures 2-3, respectively, for the linear scan and the hash lookup model. When the space budget is 0%, all methods are reduced to NoInd, and when it is 160%, all are reduced to AllInd.

It can be seen that Greedy is always the best in terms of both processing time and number of operations. DP is usually the second best one. Its performance is very close to Greedy when B_0 is small, but deteriorates when B_0 increases. The reason is that, to make DP work in practice, we have scaled B_0 down to $B_{\max}$ (as discussed at the end of Section 4.2), so the larger B_0 is, the more precision loss we have. Naive is worst than DP when B_0 is small and better than DP (still much worse than Greedy) when B_0 is large.

From this experiment, we can see that, using only 10% additional space, Greedy reduces the processing time downto 4.32% of NoInd in the linear scan model (*9.93%, in the hash lookup model*), while DP reduces to 7% (*25.36%*) and Naive reduces to 14.13% (*32.64%*). The query processing performance of Greedy is very close to the best we can do: with the all $\mathcal{R}(t)$'s materialized, AllInd improves performance to 3.36% (*1.87%*) but consumes 160% additional space.

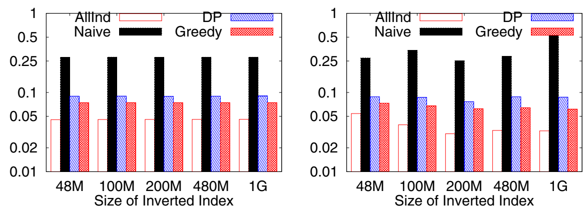
In the rest part, we fix space budget B_0 to be 10% of the size of inverted index. So Naive, DP, and Greedy all use 10% additional space below. And when the two models show similar trends, we will only report results for linear scan model.

Exp-II: Handling future queries. We then fix the query workload for index optimization to be $\mathcal{Q}_0$, and use later query workloads $\mathcal{Q}_1, \ldots, \mathcal{Q}_6$ to report processing time. In other words, at the time of index optimization, we do not know what queries will come in the future. The purpose of this experiment is to show that the performance of our index optimization techniques are stable when handling future queries.

As shown in Figures 4-5, the performance of our techniques to optimize the index is very stable. Greedy is usually very close to the best possible one AllInd (note AllInd needs 160% additional space, while others needs only 10%). In fact, the large workload of queries $\mathcal{Q}_0$ could be a good approximation to the distribution of future queries for a long time, and this is why our index optimization techniques perform well on future queries although they are based on historical queries. Fortunately, such a large query workload is usually available in practice. In Figure 5(b) we note some variation of processing time from May to Jul, which may indicate changes of the query distribution during those three months.

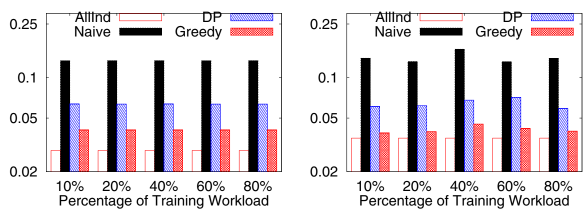
Exp-III: Varying size of inverted index. We use the 1GB inverted index and its subsets with sizes from 480MB to 48MB (randomly pick subsets of documents). After randomly partitioning $\mathcal{Q}_0$, we use 90% queries to optimize index and the rest 10% to test the performance of the index optimized using different techniques. The results are reported in Figure 6. The performance of our index optimization techniques (i.e., the improvement over NoInd) are almost unchanged in datasets of different sizes, and Greedy, which uses 10% additional space, is always very close to AllInd, which uses 150%-200% additional space. Results for the hash lookup model are similar.

Exp-IV: Comparing linear scan processing model and hash lookup model. We report the actual average query processing time for the linear scan model and the hash lookup model for comparison. With the same setting as Exp-III, we focus on only 480MB and 1GB invert indexes. The results are reported in Figure 7. Similar to results in [3] (on list intersection), the linear scan model is a bit better than the hash look up model. In both models, Greedy is always the best and shows significant improvement over NoInd, using only additional 10% memory.



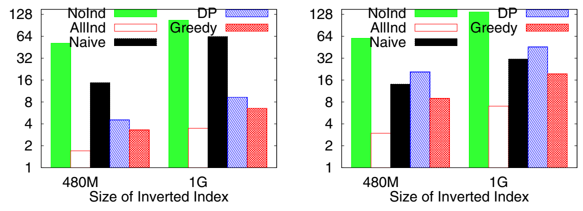
(a) # Operations (ratio) (b) Processing time (ratio)

Figure 6: Linear scan model: varying size of dataset



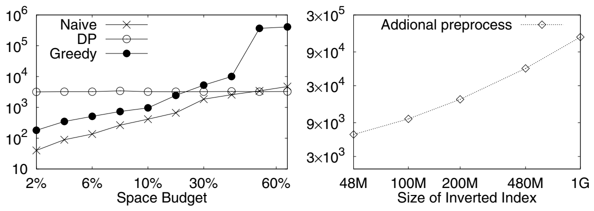
(a) # Operations (ratio) (b) Processing time (ratio)

Figure 8: Linear scan model: varying the amount of queries used for index optimization



(a) Linear scan model (b) Hash lookup model

Figure 7: Average processing time (ms) per query



(a) Optimization time (ms) (b) Construction time (ms)

Figure 9: Index optimization and construction

Exp-V: Varying amount of queries for optimization. We first randomly partition $\mathcal{Q}_0$ into ten groups. Fixing one group for testing the performance of index, we will use another different 1, 2, 4, 6, or 8 groups for optimizing the index. The results are shown in Figure 8. We can observe a slight improvement of Greedy when using 80% queries for optimization over its own performance when using less queries. This indicates the more queries used for optimization, the better.

Exp-VI: Cost of index optimization and construction. Here we report the time needed to select $\mathbf{P}$ by Naive, DP (Algorithm 2), and Greedy (Algorithm 3) in Figure 9(a) – it is mainly affected by the space budget. The time needed to construct $\mathcal{R}(t)$'s (for $t \in \mathbf{P}$) is mainly determined by the size of the inverted index, so fixing $\mathbf{P}$, we report the additional construction time in Figure 9(b). We focus on the linear scan model and trends in the hash lookup model are similar. As discussed at the end of Section 4.3, our GreedySelect in Algorithm 3 enables early termination, but will use more and more time when B_0 increases. This can be found in Figure 9(a). Once $\mathbf{P}$ is selected, the cost for constructing $\mathcal{R}(t)$'s for $t \in \mathbf{P}$ is just linear to the size of inverted index. The performance of both is quite affordable, since offline index reorganization is much less frequent than online query processing.

6. DISCUSSION AND EXTENSIONS

Handling general taxonomy

The major part of this paper focuses on a taxonomy tree. However, we note that our processing model for taxonomy queries and the index optimization schema can be also used when the taxonomy is a DAG or even contains synonym relationship. In those cases, the dynamic programming method in Algorithm 2 does not work any more, but the GreedySelect algorithm, i.e., Algorithm 3 still works, with the same approximation ratio. This is because we can prove that $\mathsf{gain}(\mathcal{Q}, \mathbf{P})$ is still a submodular function and monotone w.r.t. $\mathbf{P}$ in general taxonomies. Further testing of the performance of the optimized index on more general taxonomies is interesting future work.

Handling updates of inverted index and query log

When the document corpus is updated, we need to update both the inverted lists $\mathcal{I}(t)$'s and the result lists $\mathcal{R}(t)$'s. Assuming the selection of $\mathbf{P}$ is fixed, these two types of lists can be updated at the same time: if a document ID is inserted to (or removed from) $\mathcal{I}(t)$, we insert it into (or remove it from) $\mathcal{R}(t')$ for all the t's ancestors t''s in the term set $\mathbf{P}$.

When the updates of inverted index and query distribution (estimated from query log) is significant enough, we may also need to update the selection of $\mathbf{P}$ for better performance of the index. When the term set $\mathbf{P}$ is re-selected, we need to re-compute the whole index structure, which could be expensive. However, from Exp-II in Section 5, we note that our index optimization techniques (i.e., the selection of $\mathbf{P}$) is quite stable for future queries. So the re-selection of $\mathbf{P}$ can be done less frequently than the updates of inverted index, *e.g.*, once per month. Determining when to update $\mathbf{P}$ could be very interesting future work.

Other union/intersection operations

Our index optimization schema can be applied with different list union and list intersection algorithms embedded into our processing model (besides the simple linear scan and hash lookup models introduced in Section 3.1). For different algorithms (*e.g.*, those discussed in [3]) and processing models, we only need to seek different cost functions which approximate their actual performance precisely enough, and, in the meantime, satisfy properties in Lemma 4 or 7. Then, our index optimization algorithms introduced in Section 4 can be used to select a good $\mathbf{P}$ for materializing result lists.

Top-k taxonomy keyword queries

We do not consider the ranking of documents returned from taxonomy queries in this work. However, the optimized index structure can be directly applied in top-k retrieval for a user-specified scoring function, if we sort documents in inverted lists and result lists according to their relevance and assume the cost of retrieving the top-k is proportional to the number of documents in the complete answer to a taxonomy query. As the strength of substitutions (*e.g.*, distance between a term and its instances) may be considered in scoring and ranking, it could be interesting future work to utilize that for further improving the index's performance.

7. RELATED WORK

Techniques introduced in this work take a taxonomy in the input to optimize the index structure and to process taxonomy queries. Such a taxonomy could be constructed manually through experts and community efforts, as in WordNet [4], Cyc [14], and Freebase. With the advantage of freshness and informativeness, automatic taxonomy construction has been extensively studied recently, for example, in [20, 22, 18, 21, 26]. WikiTaxonomy [18] and YAGO [22] may be the most notable efforts, which attempt to derive a taxonomy from Wikipedia categories. With more web data, Probase [26] aims at building a unified taxonomy of worldly facts.

To the best of our knowledge, the concept of *taxonomy keyword queries* proposed in our work is new. Based on similar philosophy, there are works on how to use posting list payloads to speedup *boolean keyword queries*, such as [15, 12, 5]. For example, given a query log and a space budget, [5] studies how to select some pairs of terms to encode their co-occurrence (*e.g.*, the list of documents containing both terms in a pair). The goal is to minimize the query processing time for this query log while the additional space needed is no more than the given budget. It is important to note that both the semantics of queries and the index optimization framework in [5] are completely different from ours (they materialize term co-occurrence for some pairs while we materialize $\mathcal{R}(t)$'s for some terms), so techniques in [5] cannot be applied to our problem, and vice versa.

The idea of materializing additional information for some terms in the taxonomy is also applied in [2]. However, [2] and our work focus on different query models. In their model, a query is *one term* in the taxonomy, together with a set of indexable predicates on token strings; and a candidate answer is any token in the documents that is connected to a substitute of the query term. Candidates are scored and the top-k are returned. In our model, a query is *a set of terms*, and a candidate answer is any document that contained at least one substitute of every term in the query.

The basic operators in our query processing model are still list union and intersection. There are a lot of works on the worst-case complexity of these two operators (refer to [3] for a review). Differently, our work aims to reduce the latency of (taxonomy) query processing *for a typical query workload* by optimizing the index. And those works can be applied as operators in our query processing model, after modification to the cost function of our index optimization.

Our index optimization problem can be also thought as a view selection problem in query processing in traditional databases: given a workload of queries ($\mathcal{Q}$ in our problem), the goal is to select a set of views ($\mathcal{R}(t)$'s) to materialize, so that the processing cost for the given query workload is minimized while the space consumption is no more than a given budget (B). The view selection problem is very hard (inapproximable if P $\neq$ NP) in general [10], but there exist efficient (approximate) solutions for several different special settings and cases, for example [8, 6], and ours.

We also note that people from IR community have studied ontology-based information retrieval model, such as [24]. Different from ours, their work mainly focus on effective retrieval model, *e.g.*, adapting the classic vector-space model [24], for precise scoring, but not on the efficiency issues.

8. CONCLUSION

We introduce a new query type, taxonomy query, for the purpose of flexible query substitution. We propose processing models for taxonomy queries, and introduce how to build an additional index (besides the inverted index) to support efficient query processing. In particular, we study the problem of how to optimize this additional index based on a workload of queries, with the goal of minimizing query processing cost, and propose algorithms with performance guarantees. Our index optimization techniques are tested using real datasets and are shown to be effective and robust.

9. REFERENCES

[1] H. Cao, D. Jiang, J. Pei, Q. He, Z. Liao, E. Chen, and H. Li. Context-aware query suggestion by mining click-through and session data. In *KDD*, 2008.

[2] S. Chakrabarti, K. Puniyani, and S. Das. Optimizing scoring functions and indexes for proximity search in type-annotated corpora. In *WWW*, 2006.

[3] B. Ding and A. C. König. Fast set intersection in memory. *PVLDB*, 4(4), 2011.

[4] C. Fellbaum. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[5] M. Fontoura, M. Gurevich, V. Josifovski, and S. Vassilvitskii. Efficiently encoding term co-occurrences in inverted indexes. In *CIKM*, 2011.

[6] N. Hanusse, S. Maabout, and R. Tofan. A view selection algorithm with performance guarantee. In *EDBT*, 2009.

[7] M. A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *COLING*, 1992.

[8] C. Heeren, H. V. Jagadish, and L. Pitt. Optimal indexing using near-minimal space. In *PODS*, 2003.

[9] R. Jones, B. Rey, O. Madani, and W. Greiner. Generating query substitutions. In *WWW*, 2006.

[10] H. J. Karloff and M. Mihail. On the complexity of the view-selection problem. In *PODS*, 1999.

[11] A. Krause and C. Guestrin. A note on the budgeted maximization of submodular functions. Technical Report CMU-CALD-05-103, 2005.

[12] R. Kumar, K. Punera, T. Suel, and S. Vassilvitskii. Top-k aggregation using intersections of ranked inputs. In *WSDM*, 2009.

[13] T. Lee, Z. Wang, H. Wang, and S. Hwang. Web scale taxonomy cleansing. In *VLDB*, 2011.

[14] D. B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Commun. ACM*, 38(11), 1995.

[15] X. Long and T. Suel. Three-level caching for efficient query processing in large web search engines. *World Wide Web*, 9(4), 2006.

[16] Q. Mei, D. Zhou, and K. W. Church. Query suggestion using hitting time. In *CIKM*, 2008.

[17] D. Metzler, S. T. Dumais, and C. Meek. Similarity measures for short segments of text. In *ECIR*, 2007.

[18] S. P. Ponzetto and M. Strube. Deriving a large-scale taxonomy from wikipedia. In *AAAI*, 2007.

[19] F. Radlinski, A. Z. Broder, P. Ciccolo, E. Gabrilovich, V. Josifovski, and L. Riedel. Optimizing relevance and revenue in ad search: a query substitution approach. In *SIGIR*, 2008.

[20] R. Snow, D. Jurafsky, and A. Y. Ng. Semantic taxonomy induction from heterogenous evidence. In *ACL*, 2006.

[21] Y. Song, H. Wang, Z. Wang, H. Li, and W. Chen. Short text conceptualization using a probabilistic knowledgebase. In *IJCAI*, 2011.

[22] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[23] M. Sviridenko. A note on maximizing a submodular set function subject to a knapsack constraint. *Oper. Res. Lett.*, 32(1), 2004.

[24] D. Vallet, M. Fernández, and P. Castells. An ontology-based information retrieval model. In *ESWC*, 2005.

[25] D. Williamson and D. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[26] W. Wu, H. Li, H. Wang, and K. Zhu. Probase: A probabilistic taxonomy for text understanding. In *SIGMOD*, 2012.

APPENDIX

A.1 How to Compute $\Delta\mathsf{gain}$

To introduce how to compute $\Delta\mathsf{gain}_{\mathbf{P}}(t')$, let $\Delta\mathsf{gain}^t_{\mathbf{P}}(t') = \mathsf{gain}(t, \mathbf{P} + t') - \mathsf{gain}(t, \mathbf{P})$. And thus, from (19) and (20), $\Delta\mathsf{gain}_{\mathbf{P}}(t') = \sum_{t\in\mathcal{T}} \Delta\mathsf{gain}^t_{\mathbf{P}}(t') \cdot w(t, \mathcal{Q})$.

The linear scan model. We have

$$\begin{aligned}\Delta\mathsf{gain}^t_{\mathbf{P}}(t') &= \mathsf{cost}_s(t,\mathbf{P}) - \mathsf{cost}_s(t, \mathbf{P}\cup\{t'\})\\ &= \sum_{x\in C(t,\mathbf{P})} |\mathcal{R}(x)| - \sum_{x'\in C(t,\mathbf{P}+t')} |\mathcal{R}(x')|\\ &+ \sum_{y\in \overline{C}(t,\mathbf{P})} |\mathcal{I}(y)| - \sum_{y'\in \overline{C}(t,\mathbf{P}+t')} |\mathcal{I}(y')|. \end{aligned} \quad (23)$$

i) If t' is not a descendant of t, we cannot benefit from materializing $\mathcal{R}(t')$ for computing $\mathcal{R}(t)$. In this case, we have $C(t,\mathbf{P}) = C(t, \mathbf{P}+t')$ and thus $\Delta\mathsf{gain}^t_{\mathbf{P}}(t') = 0$.

ii) If t' is a descendant of t, but $\exists z \in \mathbf{P}$ s.t. $t' \prec z \preceq t$, then similar to i), we have $\Delta\mathsf{gain}^t_{\mathbf{P}}(t') = 0$.

iii) If t' is a descendant of t and $\nexists z \in \mathbf{P}$ s.t. $t' \prec z \preceq t$, then we have $C(t,\mathbf{P}) - C(t,\mathbf{P}+t') = C(t',\mathbf{P})$ and $C(t,\mathbf{P}+t') - C(t,\mathbf{P}) = \{t'\}$. So in this case,

$$\Delta\mathsf{gain}^t_{\mathbf{P}}(t') = \Delta\mathsf{gain}^{t'}_{\mathbf{P}}(t'). \quad (24)$$

In a special case, if $\mathbf{P}$ does NOT contain any descendant of t', then $C(t,\mathbf{P}) = \varnothing$ and $C(t,\mathbf{P}+t') = \{t'\}$, so

$$\Delta\mathsf{gain}^t_{\mathbf{P}}(t') = -|\mathcal{R}(t')| + \sum_{y\in\mathcal{S}(t')} |\mathcal{I}(y)|. \quad (25)$$

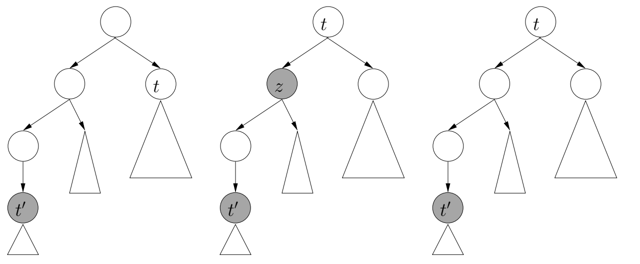
From cases i)-iii), we have

$$\Delta\mathsf{gain}_{\mathbf{P}}(t') = \sum_{t:\ t'\preceq t\prec\mathcal{A}(t')\cap\mathbf{P}} \Delta\mathsf{gain}^t_{\mathbf{P}}(t')\cdot w(t,\mathcal{Q}), \quad (26)$$

where $t \prec \mathcal{A}(t')\cap\mathbf{P}$ means that $t \prec x$ for any $x \in \mathcal{A}(t')\cap\mathbf{P}$ (if $\mathcal{A}(t')\cap\mathbf{P} = \varnothing$, it is true). Note that $t' \preceq t \prec \mathcal{A}(t')\cap\mathbf{P}$ is essentially equivalent to the condition of the case iii).

In the special case where $\mathbf{P}$ does NOT contain any descendant of t', $\Delta\mathsf{gain}^t_{\mathbf{P}}(t')$ can be computed as (25). Let $S = \mathcal{A}(t')\cap\mathbf{P}$. The value $\Delta\mathsf{gain}_{\mathbf{P}}(t')$ is determined by S and t'. So we can write $\Delta\mathsf{gain}_{\mathbf{P}}(t')$ as $\Delta\mathsf{gain}_S(t')$.

Lemma 4 follows from the above discussion, because from the numbering of nodes in $\mathcal{T}$, $\mathbf{P}'$ and $\mathbf{P}''$ do NOT contain any descendant of t_i. We can compute the r.h.s. of (25) for each term t' in preprocessing, and then computing $\Delta\mathsf{gain}_S(t')$ as (26) needs $O(h)$ time, where h is the height of the taxonomy tree, when $\Delta\mathsf{gain}_S(t')$'s value is needed in Algorithm 2.



(a) Case i) (b) Case ii) (c) Case iii)

Figure 10: Cases when computing $\Delta\mathsf{gain}^t_{\mathbf{P}}(t')$ (white nodes are not in $\mathbf{P}$, and gray ones are in $\mathbf{P}+t'$)

To compute $\Delta\mathsf{gain}_{\mathbf{P}}(t)$ in $O(h)$ time in Algorithm 3, we need to maintain $\Delta\mathsf{gain}^{t'}_{\mathbf{P}}(t')$ for each term t'. As $\mathbf{P}$ is updated with one more term added, there are $O(h)$ terms t''s whose $\Delta\mathsf{gain}^{t'}_{\mathbf{P}}(t')$'s need to be updated. When $\Delta\mathsf{gain}_{\mathbf{P}}(t)$, is needed, it can be computed as (24) and (26) in $O(h)$ time.

The hash lookup model. In this model, everything stays unchanged except that (23) and (25) are replaced with:

$$\begin{aligned}\Delta\mathsf{gain}^t_{\mathbf{P}}(t') &= \mathsf{cost}_h(t,\mathbf{P}) - \mathsf{cost}_h(t,\mathbf{P}\cup\{t'\}) \quad (27)\\ &=|C(t,\mathbf{P})| - |C(t,\mathbf{P}+t')| + |\overline{C}(t,\mathbf{P})| - |\overline{C}(t,\mathbf{P}+t')|.\\ \text{and}\quad \Delta\mathsf{gain}^t_{\mathbf{P}}(t') &= -1 + |\mathcal{S}(t')|, \quad (28)\end{aligned}$$

respectively. And all the discussion stays the same.

A.2 Submodularity of Function gain

To prove that $\mathsf{gain}(t,\mathbf{P})$ w.r.t. $\mathbf{P}$ is submodular for any fixed t, we only need to prove that, for any $x, t' \notin \mathbf{P}$ s.t. $x \neq t'$, we have $\Delta\mathsf{gain}^t_{\mathbf{P}+x}(t') \leq \Delta\mathsf{gain}^t_{\mathbf{P}}(t')$.

Now let us re-examine the cases i)-iii) as in A.1.

i') If t' is not a descendant of t, from the analysis of i), we already have $\Delta\mathsf{gain}^t_{\mathbf{P}+x}(t') = \Delta\mathsf{gain}^t_{\mathbf{P}}(t') = 0$.

ii') If t' is a descendant of t, but there exists $z \in \mathbf{P} \subseteq \mathbf{P}+x$ s.t. $t' \prec z \preceq t$, then from the analysis of ii), we also have $\Delta\mathsf{gain}^t_{\mathbf{P}+x}(t') = \Delta\mathsf{gain}^t_{\mathbf{P}}(t') = 0$.

iii') If t' is a descendant of t and there does NOT exist $z \in P$ s.t. $t' \prec z \preceq t$ but $t' \prec x \preceq t$, then from ii), $\Delta\mathsf{gain}^t_{\mathbf{P}+x}(t') = 0$, and from iii), $\Delta\mathsf{gain}^t_{\mathbf{P}}(t') \geq 0$.

iii") The rest case is: t' is a descendant of t and there does NOT exist $z \in P + x$ s.t. $t' \prec z \preceq t$. In this case, considering iii) and (24), we only need to prove

$$\Delta\mathsf{gain}^{t'}_{\mathbf{P}+x}(t') \leq \Delta\mathsf{gain}^{t'}_{\mathbf{P}}(t').$$

In fact, from (23), in the liner scan model, we have

$$\begin{aligned}\Delta\mathsf{gain}^{t'}_{\mathbf{P}+x}(t') &= -|\mathcal{R}(t')| + \mathsf{cost}_s(t',\mathbf{P}+x)\\ &\leq -|\mathcal{R}(t')| + \mathsf{cost}_s(t',\mathbf{P}) = \Delta\mathsf{gain}^{t'}_{\mathbf{P}}(t').\end{aligned}$$

And similarly, in the hash lookup model, we have

$$\begin{aligned}\Delta\mathsf{gain}^{t'}_{\mathbf{P}+x}(t') &= -1 + \mathcal{N}(t',\mathbf{P}+x)\\ &\leq -1 + \mathcal{N}(t',\mathbf{P}) = \Delta\mathsf{gain}^{t'}_{\mathbf{P}}(t').\end{aligned}$$

From the above four cases, we can conclude that $\mathsf{gain}(t,\mathbf{P})$ w.r.t. $\mathbf{P}$ is submodular for any fixed t. Since $\mathsf{gain}(\mathcal{Q},\mathbf{P})$ is the weighted sum of $\mathsf{gain}(t,\mathbf{P})$'s, it is also submodular.