Relational Databases for Querying Natural Language Text

Pirooz Chubak
Department of Computing Science
University of Alberta
pchubak@cs.ualberta.ca

Davood Rafiei
Department of Computing Science
University of Alberta
drafiei@cs.ualberta.ca

Abstract

With the vast amount of information stored in natural language text, sophisticated query engines are needed to pull data and effectively relate the pieces. While there has been a great deal of activity around semistructured data and in particular XML, there hasn't been much work on querying natural language text, despite the regularities that exist in natural language text. This paper explores a more conservative approach where natural language text is stored in a relational database. We present a framework for querying and integrating natural language text with relational data and investigate different strategies for optimizing queries. Our results show that the size of the plan space depends on the number of query terms and the overlap between query rewritings. Moreover, we show that the complexity of finding an optimal plan in the presence of rewritings is NP-hard. We develop a cost model and pruning techniques to reduce the size of the search space, and a polynomial-time greedy algorithm that finds a sub-optimal plan over a set of rewritings. Our experimental results indicate great savings in the evaluation costs of the optimized queries and that our greedy algorithm finds either an optimal plan or a plan that is very close to optimal in terms of cost.

1 Introduction

There is a large volume of facts expressed in natural language text, and we often want to extract these facts, relate them with other facts, or ask more specific questions. Examples include extracting the list of genes mentioned in a collection of biomedical literature, finding if a paper has experimental evidence for gene products [2], gathering opinions on a topic or a product from forums and newsgroups, etc. Past work in this area is usually either specific to a particular domain or task (e.g. named-entity recognition [9]) or assumes a clean and relatively small text collection such as news corpora [20]. There are recent attempts to scale up fact extraction to large collections such as the Web with some limited success (e.g. [16]). However, to the best of our knowledge, there is little work on more general approaches for querying natural language text and evaluation strategies that can scale up to very large text corpora. The problem is challenging because natural language text has little structure (compared to relational standards or even XML). However natural language text is also richer than pure text in terms of the way facts are organized; there are rules and regularities governing natural language text that can be exploited by a querying engine. Also term frequencies and the co-occurrence statistics are meaningful.

In this paper, we study the scenario where queries over text are expressed in *Natural Language Text Queries* (NLTQ) and a rewriting engine is used for query expansion [14]. The syntax and the semantics of the queries and the rewritings are briefly discussed in Section 3. We further assume that NLTQ queries are integrated into SQL, allowing data from both text and relational sources to be integrated in a query.

1.1 A Motivating Example

Suppose we have partial lists of genes and syndromes, and we want to search for more genes, syndromes and possible relationships that may have been reported in a text collection such as Medline [1]. Suppose we are interested in casual relationships between gene defects and syndromes. In a typical setting, the set of known genes and syndromes may be stored in relational tables and the query may be written as follows:

```
(SELECT x, y FROM genes g,
"%x gene defects responsible for %y"
on medline WHERE g.name=x)
UNION
(SELECT x, y FROM syndromes s,
"%x gene defects responsible for %y"
on medline WHERE s.name=y)
```

`"%x gene defects responsible for %y" on medline` is expected to return pairs of genes and syndromes that are reported to have the relationship we are seeking. It is feasible that the texts of queries and data may not exactly match. For example, the query does not exactly match "the X-linked form is a result of mutations in the CD40 ligand gene". A rewriting engine can expand the query into alternative expressions such as `"%x mutations in %y"` and `"%y is a result of mutations in %x gene"`. Paraphrasing is an

effective method for increasing query recalls over natural language text.

1.2 The Problem Statement and Our Contributions

In this paper, we take a more conservative approach and assume that natural language text is stored in a relational database. This has the benefit that data from both text and relations can be joined in queries, and relational engine functionalities can be exploited for expressing queries and query optimization. The problem to be addressed is given a SQL query with NLTQ expressions, for example in the *from clause*, we want to map the query to an equivalent query or query plan that can be efficiently evaluated by a SQL engine. Given that relational query optimization is a well-studied subject, our focus in this paper is on mapping and optimizing NLTQ expressions. Our cost models and estimates are based on the same statistics that are typically available to a relational optimizer, hence our methods can potentially be integrated into a relational engine.

Our first contribution is a cost model for estimating the efficiency of a mapping, in terms of the expected number of I/Os, and our experimental results on the accuracy of our estimations. As the second contribution, we develop strategies for pruning query plans that are guaranteed not to be optimal; therefore the size of the search space for an optimal plan is significantly reduced. Our third contribution is on optimizing query rewritings. Given the set of rewritings of a query, some overlap is expected between the terms of the query and its rewritings and also between the terms of the rewritings. In the presence of an overlap, independently optimizing each rewriting is not guaranteed to give the overall best plan. We formalize the search for an optimal plan for a set of rewritings as an optimization problem and derive analytical results on its complexity; our results show that the problem is at least NP-hard. As our last contribution, we relax our optimality criterion and develop an efficient greedy algorithm for finding sub-optimal plans. Our experimental results show the greedy algorithm finds either an optimal or a near-optimal plan at a much lower cost.

2 Related Work

Related work can be grouped into 6 categories: (a) query optimization over text, (b) integration of relational data and text, (c) multi-query optimization, (d) full-text support in commercial databases, (e) named entity recognition and question answering, and (f) extracting relations from text.

Query optimization over text There is work on querying and optimizing queries over text and semi-structured data but not specifically on natural language text. PAT [18] for example, is a system for searching text with some commercial success [11]; It introduces text regions as first class citizens. The algebra behind PAT is studied more closely by Consens and Milo where they show the relationship between region algebra and Monadic first order theory of binary trees and the complexity of optimizing queries in the algebra [10]. Unlike PAT, NLTQ is restricted to natural language text and somewhat makes use of the structure of the sentences.

Integration of relational data and text Chaudhuri, Dayal and Yan [7] study several techniques for joining relational data with external text sources. The join methods include semi-join, tuple substitution and probing. The authors show that the best performing method varies with the selectivity and the fraction of joining tuples. Unlike this work which treats text system as a black box, our work optimizes text queries based on their declarative expressions. Also, since text data is stored in relational tables, a "tight" join between text and relational tables is feasible. The work on estimating the selectivity of text predicates [8] is also related to ours and may be used in estimating the selectivity of our text queries after being mapped to SQL.

The works on combined querying of text documents and relations (such as WSQ/DSQ [13] and [12]) are also related. Unlike ours, the result of a text search is usually a set of matching documents here.

Multi-query optimization Roy et al. [17] study the optimization of multiple queries. Their method benefits from materializing and re-using common sub-expressions that exist among different queries. They model their optimization problem with Directed Acyclic Graph (DAG) representation and introduce heuristics for improving the performance of multi-query evaluation. Our work on optimizing the evaluation of multiple rewritings in section 6 is similar to [17] but has a few important differences. First, given a query (or a rewriting) in our scheme, any subset of the query terms can be used for filtering and for each subset, there is a different set of possible plans. This is unlike the queries in Roy et al. where the query terms or relations are fixed for each query. Second, the search space for the best plan is the union of the plans for all possible term subsets. In the presence of multiple rewritings, the search space is the Cartesian product of the plan sets for different rewritings. We are not sure if a DAG in the style of Roy et al. can be constructed or would be effective for this search space. Third, our greedy algorithm is similar to the one by Roy et al., with a difference that ours enumerates term overlaps whereas theirs iterate over DAG nodes.

Full-text support in commercial databases Many commercial database management systems support facilities for full-text search functionalities integrated into their relational engine (See DB2 Text Extender[15] and Oracle InterMedia Text[3]). The output of a text search again is a set of matching documents and it is not easy to tightly integrate and interrelate them with relational data.

Named entity recognition and question answering Related work also includes the literature on named entity recognition where given a fixed set of categories such as

person names, locations, percentages and monetary values, the task is to extract and classify the elements in text to one of those categories [9]. There is also work on question answering where given a natural language question, the task is to find the most relevant answer from a text collection [20]. Our work here is different in that the set of types is not fixed in advance and can vary with queries.

Extracting relations from text Finally our work is related to the literature on extracting relations from text [19] and web pages [6, 4]. Our work is based on natural language queries of DeWild [14] which have a more confined syntax, allowing us to map text queries to SQL.

3 Problem Formulation

This section presents the syntax and the semantics of NLTQ and our mapping of NLTQ expressions to query plans over relations that store natural language text.

3.1 Mapping Natural Language Text to Relations

Definition 1. *A natural language text query (NLTQ) is a sequence of terms, phrases and wild cards. Each NLTQ must have at least one extractor wild card and can be represented with the following grammar.*

```
<PHRS> ::= term | <PHRS> term
<WLCD> ::= %variable | *<PHRS>*
<NLTQ> ::= %variable
<NLTQ> ::= <PHRS> <NLTQ> | <NLTQ> <PHRS>
<NLTQ> ::= <WLCD> <NLTQ> | <NLTQ> <WLCD>
```

The extractor wild card, denoted by `%variable`, can replace a noun or a noun phrase. The query syntax includes another wild card, denoted by `*<PHRS>*`, for query expansion. The wild card indicates that the enclosed phrases can be replaced with similar terms and phrases without much affecting the meaning of the query. The result of a query on a text collection is a table with one column for each extractor and includes all assignments of the variables that give rise to a match.

Definition 2. *A natural language text query q' is a rewriting of query q if both q and q' have the same extractors and every match for q' is also considered a match for q.*

As an example, the query `%x is the *author* of %y` extracts pairs of *x* and *y*, where *x* is an author of *y*. Since 'author' is enclosed in *'s, similar terms to 'author' are considered and the query is re-evaluated. For the given query, similar terms to 'author' can be 'writer' and 'co-author'. Moreover, a set of query rewritings for the given query could be ``%x, author of %y'', ``%y written by %x'' and ``%x wrote %y''.

In this paper, we take a relational database approach to query optimization. Input text is stored in two tables: `terms(term, docid, sid, offset, length, pos)` and `sentences (sentence, sid, docid)`. The key for sentences is `docid, sid` and the same set of attributes is a foreign key in terms referring sentences. There are indexes on `term, docid, sid, offset` and on `sid, docid, term` of terms and on `sid, docid` of sentences, and we assume partial match searches are also supported which is the case in most commercial relational databases. Note that all parsings and speech taggings necessary are done by dedicated NLP tools before the text is stored in relations. A study of these preprocessings are outside the scope of this work.

The choice of the schema is largely influenced by the syntax and semantics of our queries. The matching boundary of a NLTQ cannot be larger than a sentence. Hence, the sentences table is sufficient to answer any NLTQ. On the other hand, the smallest unit that can match a wild card is a term. Also, the terms table allows us to do IR-style inverted-list pruning for our queries. Table of n-grams and phrases may also be beneficial for some queries (e.g. [5]); but since they are less general, they are not considered here.

Given a NLTQ, the query is mapped into an execution plan over the base tables. In the style of relational query optimizers, a query plan is best described as a tree with base tables at the leaves and the operations at the intermediate nodes or edges.

3.2 Query Optimization Problem

The space of possible plans for a query Q in general is exponential on the number of query terms as shown in Section 5. Filtering sentences based on all query terms is not necessarily the best strategy since each filtering also introduces an overhead. Other plan trees are possible by changing the order of the selections and considering other tree structures such as bushy trees. The space of possible plans is even larger if we consider rewritings and the overlaps between their query plans. For instance, there are terms that appear in multiple rewritings and a query optimizer should be aware of such overlaps in enumerating the plans and cost estimations. We are interested in plans with minimal expected costs. The problem to be addressed is: *given a query Q and its set of rewritings $R(Q)$, find the "best" evaluation plan*. Here the "best" refers to the plan with the least estimated I/O cost, according to our cost model to be discussed next. The right choice of a plan can have a great influence on the cost of query evaluation as shown in some of our experiments.

4 Cost Model

Given a query plan, its cost can be estimated in terms of the expected number of I/Os. Each node in the query plan tree can be considered as the root of a subplan tree; the evaluation cost for each node is the sum of the costs of evaluating its children and the cost of joining the results.

4.1 Join Cost Estimations

Given nodes n_1 and n_2, the join conditions are $n_1.sid = n_2.sid$ and $n_1.docid = n_2.docid$. Let $l(n)$ and $r(n)$ respectively give the left child and the right child of node n.

$$c_1(n) = \begin{cases} C_a + f_{low}\left(\frac{C_t}{C_p} + C_a\right) & l(n), r(n) = leaves \\ f_{low} \cdot C_a & otherwise \end{cases}$$

$$c_2(n) = \begin{cases} 2C_a + \frac{C_t}{C_p}(f_{low} + f_{high}) & l(n), r(n) = leaves \\ C_a + \frac{C_t}{C_p} \cdot f_{high} & otherwise \end{cases}$$

Where $f_{low} = \min\{f(l(n)), f(r(n))\}$ and $f_{high} = \max\{f(l(n)), f(r(n)))\}$. c_1 models the cost of an index nested loop join, whereas c_2 models the scenario where all the results from both left and right subtrees are retrieved before a join. In our cost models, C_a is the cost of retrieving the first page from an index with matching entries for a given query. In a typical setting, we can assume that C_a is equal to 1.2 I/Os on average [1]. C_t is the size of an index entry in bytes, and C_p is the page size. Therefore, $\frac{C_t}{C_p}$ gives the fraction of a page that is occupied by a single index entry. $f(n)$ is the size of the result set at node n, in terms of the number of distinct sentences that are retrieved. Assuming that the terms occur independently in sentences [2], the size of the result set at a node n is given by:

$$f(n) = \begin{cases} S_n & n = leaf \\ \frac{f(l(n)) \times f(r(n))}{S_t} & otherwise \end{cases}$$

In which S_n is the number of sentences that contain the term at node n and S_t is the total number of sentences.

4.2 Estimating the Cost of a Plan

The cost of evaluating a plan rooted at node n is defined as the sum of the costs of evaluating the left and the right subtrees, the cost of the join and the cost of storing and retrieving any intermediate results (if needed). More formally, the cost can be recursively defined as follows:

$$c(n) = \begin{cases} 0 & n = leaf \\ \min\{c_1(n), c_2(n)\} + & \\ c(l(n)) + c(r(n)) + c_s(n) & n \neq leaf \end{cases}$$

where $c_s(n)$ is the cost of storing and retrieving any intermediate results. When evaluating a non-leaf node, we sometimes cannot directly pipe the intermediate results. Thus, we need to store the results of the left or the right subtree on the secondary storage before evaluating their join. The cost of writing the result set of a node n directly depends on the size of its result set and is $c_w(n) = (C_t/C_p)f(n)$. Reading from disk is usually a little bit faster than writing on disk. However, since costs of reading and writing are small compared to join costs, we can make the simplifying assumption that $c_r(n) = c_w(n)$. Having this assumption, C_s can be computed as follows:

$$c_s(n) = \begin{cases} 0 & l(n) \vee r(n) = leaf \\ c_r(l(n)) + c_w(l(n)) & f(l(n)) < f(r(n)) \\ c_r(r(n)) + c_w(r(n)) & f(l(n)) \geq f(r(n)) \end{cases}$$

Finally, the qualifying matching tuples must be joined with sentences. Assuming that $r(n)$ is the *sentences* table, the total cost c_t at the root n can be given as

$$c_t(n) = c(l(n)) + c_1(n).$$

It might be desirable to confine the result set of the query to terms with specific part of speech tags. For example we would only like to have nouns in the result set. This could be done by a final join of the result set on the `pos` column of terms table. Since this additional cost does not change our optimization results, we do not include it in our optimization model for brevity.

4.3 Term Associations

Although assuming independence for terms in a sentence somewhat simplifies the cost estimates, it is not hard to list many cases where this assumption fails.

Definition 3. *Term association score is a number between 0 and 1 which describes the confidence that two terms occur in the same sentence relative to their expected co-occurrence value when the terms are assumed to be independent. For the nodes n_1 and n_2, term association is defined as:*

$$a(n_1, n_2) = \begin{cases} 0 & f(n_1, n_2) < \frac{f(n_1)f(n_2)}{S_t} \\ \frac{f(n_1,n_2) - \frac{f(n_1)f(n_2)}{S_t}}{\min(f(n_1), f(n_2))} & f(n_1, n_2) \geq \frac{f(n_1)f(n_2)}{S_t} \end{cases}$$

where $f(n_1, n_2)$ is the joint frequency of two terms and is defined as the number of sentences that contain the terms at nodes n_1 and n_2. Note that n_1 and n_2 must both be leaves, otherwise the term association will be undefined. If the expected frequency and real frequencies are equal, association will be zero. On the other hand the higher the association, the more we are underestimating the expected joint frequency.

Storing association scores for all term pairs can be costly. For example, a text collection we have been experimenting with had 64,783 unique terms and 8.3 million association pairs with nonzero joint frequency (occuring together at least once in a sentence). To reduce the size, one heuristic is to remove the pairs whose associations don't have a significant effect on the joint frequency. These would include the entries with an association score less than a threshold. As Figure 1 suggests for our dataset, with a threshold of 0.2, we can reduce the number of entries to almost 10% of its previous size, leaving less than 850,000 term pairs in the association table. Finally, to remove the noisy and meaningless terms that have been seen very infrequently, we define a *support* factor for the minimum frequency of terms. This leaves only around 26,000 tuples in the association table when the support factor is 5. With that many

[1] This is a conservative estimate for a B+-tree assuming that the first few levels of the index are cached.

[2] It should be noted that in a more realistic setting, terms that appear in a sentence are not independent. For example, the terms of compound words and phrases are more likely to appear together. Section 4.3 discusses this issue in more details.

pairs, the association table can be cached by the query optimizer for fast look-ups. Note that we only maintain the scores of frequent pairs and as figure 1 suggests the number of those pairs drops exponentially as the association score increases. Therefore, we expect an association table to be scalable for large text collections. If the terms at nodes n_1 and n_2 have an entry in the association table, their joint frequency is given by

$$f_a(n_1, n_2) = a(n_1, n_2) \min(f(n_1), f(n_2)) + \frac{f(n_1) f(n_2)}{S_t}.$$

Otherwise, the terms can be treated independently. In order to generalize the above formula, we need to check all the term pairs in a query sub-tree for which we are estimating the joint frequency. Suppose we would like to join a subtree T_1 with another subtree T_2 and estimate their joint frequency. Given the estimated frequency for T_1 and T_2, the joint frequency can be estimated as follows:

$$f_a(T_1, T_2) = f(T_1) f(T_2) \prod_{n_1 \in T_1} \prod_{n_2 \in T_2} \frac{f_a(n_1, n_2)}{f(n_1) f(n_2)}$$

where n_1 and n_2 represent nodes in subtrees T_1 and T_2 respectively.

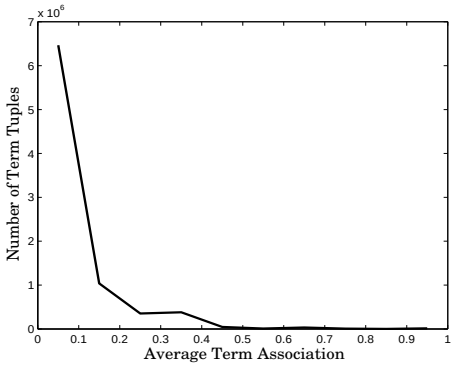

Figure 1. Distribution of term associations

5 Query Plan Optimization

Given a query with N terms, a query plan can choose any combination of the terms placed at the leaves of a plan tree.

Lemma 1. *The number of query plans for a NLTQ with N terms is given by $\sum_{n=1}^{N} qss(N, n)$ where*

$$qss(N, n) = \begin{cases} N & n = 1 \\ \frac{1}{2} \sum_{i=1}^{n-1} qss(N, i) qss(N - i, n - i) & n > 1. \end{cases}$$

Proof. Appears in the appendix. □

The number of query plans in general can be huge, and searching the plan space for a plan with the least cost can be computational-intensive. We want to reduce the size of the search space while still keeping the plan with the least cost in the reduced space. The following theorems will show that there exists a linear solution that finds the optimal plan for a single query.

Theorem 1. *Assuming terms independence, for a query plan with n leaves represented in a Left Deep Tree (LDT), the lowest estimated cost can always be obtained by sorting terms according to their frequencies and placing lower frequency terms on leaves in higher depths.*

Proof. Appears in the appendix. □

Definition 4. *We say two binary trees t_1 and t_2 are traversal equivalent when there exists a Depth First Search (DFS) traversal that produces the same leaf sequence on both t_1 and t_2. We denote a traversal equivalence by $\cong$.*

As an example, the following two tree representations are traversal equivalent, i.e. (((a b) c) (d e)) $\cong$ ((((a b) c) d) e). Since both trees only have labels at the leaves, all the DFS traversals produce the same leaf sequence.

Theorem 2. *Assuming terms independence, any query plan t with n leaves has a traversal equivalent left deep tree that always has an estimated cost less than or equal to the estimated cost of t.*

Proof. Appears in the appendix. □

Based on Theorems 1 and 2, we know that an optimal plan is always a left-deep tree with terms sorted by their frequencies and placed at the leaves with lower frequency terms in higher depths. The number of those plans cannot exceed N. A linear search over that many plans is guaranteed to find an optimal plan. We refer to an optimal plan for a single rewriting as a local optimal plan.

6 Optimization Over Multiple Rewritings

In the presence of multiple rewritings, finding an optimal plan for each rewriting is not guaranteed to give an overall plan with the total least cost. In particular, if the plan trees of two or more rewritings share the same subplans, it might be cheaper to evaluate the shared subplans only once and feed their results to the plans.

6.1 Overlap Handling

The main idea for optimizing a set of rewritings is to take advantage of common terms in their query expressions. If there are two or more terms that are shared among multiple queries, it may worth to isolate the terms into a subplan and evaluate and store the result for future uses. These subplans can be evaluated once and used many times. We use the term subplan to refer to a part of a plan that may be shared between more than one rewriting. To find a plan with the least total cost over a set of rewritings, we would need to first find a set of subplans that are shared by multiple rewritings and are also worth materializing.

As an example, consider a query that would give a list of athletes who achieved a gold medal in any of the Olympics from a corpus of archived data on sports news. Suppose the query to evaluate is `R0:%x, an Olympics gold medalist` and its set of rewritings is given as follows.

```
R1:%x, the Olympics gold medalist
R2:%x won an Olympics gold medal
R3:%x was a champion in Olympics
R4:%x stood first in Olympics
R5:the gold medal of Olympics was given to %x
```

The following table gives the plan with the minimum cost for `R0` and each of its rewritings when each query is optimized individually; the third column of the table shows best plan costs, in terms of the estimated number of I/Os. We refer to these plans as *local optimal* or *best local* plans.

Rewriting	Best Local Plan	Cost
R0	(Olympics medalist)	29.2
R1	(Olympics medalist)	29.2
R2	(won Olympics)	26.2
R3	(champion Olympics)	20.2
R4	(stood Olympics)	21.4
R5	(Olympics given)	31.8

A set of subplans that are shared between at least two of the above query rewritings are given in the following table. The last column in this table shows the amount of saving in terms of the reduction in the estimated number of I/Os if the subplan is materialized. The saving is measured over the sum of the costs of local optimal plans.

Subplan	Rewritings	Saving
B0:((Olympics medalist) gold)	R0,R1	26.4
B1:((Olympics medal) gold)	R2,R5	10.9
B2:(Olympics was)	R3,R5	-1.3
B3:(Olympics gold)	R0,R1,R2,R5	-46.9
B4:((Olympics gold) an)	R0,R2	-79.4
B5:((Olympics gold) the)	R1,R5	-98.2
B6:(Olympics in)	R3,R4	-598.7

Deciding which subplans to materialize, and which rewritings should use the materialized subplans can be tricky. Selecting a subplan to be materialized can influence the cost of other subplans. The cost function for a given set of rewritings is not necessarily linear, if the subplans are not independent; i.e. for two interdependent subplans B_1 and B_2 the total cost when both subplans are materialized is not the sum of the costs when each subplan is materialized.

The problem to be addressed is to find a set of plans, one for each rewriting, and a set of subplans, materialized in advance and reused in plans, such that the total cost of evaluating the plans and materializing the subplans is minimal. In Section 6.3 we introduce a heuristic and a greedy algorithm that selects the subplans according to their estimated cost saving. Once a subplan is selected to be materialized, the other subplans can choose to use it to reduce their own cost. The next section analyzes the complexity of problem in its general form.

6.2 Complexity Analysis

Given a query q, let $R(q)$ be the collection of its rewritings including the query and $T = \cup_{r_i \in R(q)} T(r_i)$.

Definition 5. *A subplan s is legal over $R(q)$ if there is $r_i \in R(q)$ such that $T(s) \subseteq T(r_i)$.*

Definition 6. *Let M denote the set of all legal subplans of $R(q)$. We call $M' \subseteq M$ an optimal materialization set over $R(q)$ if the sum of the costs of materializing M' and evaluating $R(q)$ with M' materialized is minimal; i.e.*

$$\sum_{m_i \in M'} C(m_i, \emptyset) + \sum_{r_j \in R(q)} C(r_j, M')$$

is minimal where $C(x, Y)$ is the cost of evaluating x given that Y is evaluated in advance and its result is materialized.

Theorem 3. *The problem of finding an optimal materialization set is NP-hard.*

Proof Consider a slightly simpler version of the problem where we want to find if there exists $M' \subseteq M$ such that

$$\sum_{m_i \in M'} C(m_i, \emptyset) + \sum_{r_j \in R(q)} C(r_j, M') \leq k$$

for a fixed k. If we show the NP-hardness for this simplified version, the proof for the more general version follows. We prove this by providing a reduction from the minimum cover problem. Given a collection B of subsets of S, a minimum cover of size k or less for S is $B' \subseteq B$ such that he union of the sets in B' is S and $|B'| \leq k$. Define a cost function C as follows: (1) $C(b_i, \emptyset) = 1$ for every $b_i \in B$, (2) $C(s_j, B') = 0$ if $B' \subseteq B$ and there exists $b_i \in B'$ that covers s_j, and (3) $C(s_j, B') = \infty$ otherwise. B' is a minimum cover of size at most k for S if

$$\sum_{b_i \in B'} C(b_i, \emptyset) + \sum_{s_j \in S} C(s_j, B') \leq k.$$

□

Having a polynomial time algorithm for finding an optimal materialization set implies that we have a polynomial time algorithm for the minimum cover which is unlikely (unless P=NP). A naive algorithm may examine all possible query plans which is expected to be large for large number of terms and rewritings. Next we give a sub-optimal algorithm that runs in polynomial time.

6.3 A Suboptimal Solution

We propose a greedy algorithm called Common Subplan Greedy (CSGreedy) that gives a suboptimal solution with a polynomial time complexity. This algorithm chooses the subplans according to their savings, and the subplan with the highest saving is chosen first. In each step, the algorithm estimates the total cost of rewritings using the subplans chosen so far, and continues until the cost is not decreasing any more. The overall suboptimal plan is the plan with the minimum total cost. Intuitively, subplans which have low costs

and are shared among a large number of rewritings, have priority to be selected.

The algorithm, as presented in Figure 2, takes a set R of rewritings and returns a suboptimal solution P and an estimated total cost for P. B is the set of subplans that would need to be materialized; it is initially empty. In steps 2-4, P is initialized to empty set and is incrementally populated with the tuples $< r, p >$ where $r \in R$ is a rewriting and p is a local optimal plan for r. The search space for finding a local optimal plan for each rewriting has a size linear to the size of the query, as discussed in Section 5.

CSGreedy(R)

```
1   B ← ∅
2   P ← ∅
3   foreach r ∈ R do
4       P ← P ∪ {< r, local_optimal_plan(r) >}
5   total_cost ← compute_total_cost(P, B)
6   subplans ← get_all_subplans(R)
7   Sort the subplans based on their savings over P such that
    saving(subplans(i)) ≥ saving(subplans(j)) iff i < j
8   foreach i = 1, ..., |subplans| do
9       B ← B ∪ subplans(i)
10      cost ← compute_total_cost(P, B)
11      if cost < total_cost then
12          total_cost ← cost
13          Update the plans in P assuming that B is materialized
        else
14          return < P, total_cost >
15  return < P, total_cost >
```

Figure 2. A greedy algorithm for finding suboptimal plans

In line 5, we find the total cost of the rewriting set which is the sum of the minimum local costs of the rewritings since B is empty. In lines 6 and 7 we find all subplans that can be built from our set of rewritings and sort the subplans according to their savings over the total cost estimated in Step 5. In lines 8-14, we iterate over the subplans, from the one with the greatest saving to the one with the least and add each subplan to B. Then we recompute the total cost of rewritings given that the set B is materialized; the cost here also includes the cost of materializing B and any additional readings that may be needed. The iteration continues until the point where materializing a subplan does not reduce the cost. The algorithm returns a suboptimal plan and its cost.

In order to compute the complexity of our algorithm, we assume that N_a is the average number of terms per rewriting and k is the total number of rewritings. For lines 3 and 4, the algorithm iterates over $\sum_{i=1}^{k} |r_i|$) operations, for which $|r_i|$ is the size of rewriting i in terms of the number of terms. Therefore, the complexity for this section of CSGreedy is approximately $\theta(k \times N_a)$. Since we find and store the costs of local optimal plans in lines 3 and 4, line 5 has a complexity no more than $\theta(k)$, which is negligible. In line 6, CSGreedy computes all subplans that are shared between two or more rewritings. Each subplan must have at least two terms before it is useful. In order to compute the complexity of the algorithm in lines 8-14 we model each term as a random variable T that may happen to be in a rewriting $R_i, i = 1..k$ with a probability $P(X \in R_i) = N_a/N$ where N is the total number of terms. Given this probability and solving for the expected number of unique overlaps between any selection of rewritings, we would get the following formula for the expected number of unique overlaps:

$$\sum_{l=2}^{k} \sum_{q=2}^{N} \frac{BinPDF(l; k, p^q) \times BinPDF(q; N, p^l)}{p^{lq}}$$

where $BinPDF$ is the binomial probability distribution function. The reason both l and q are initiated to 2 and greater is that we would like at least two terms be shared with at least two rewritings, otherwise we would not consider them for caching. Finally, we make a simplifying assumption that the number of terms is greater than the number of rewritings, which is the case in most of our rewriting sets. A numerical analysis shows that the expected number of unique overlaps is approximately $O(N_a k^{\frac{3}{2}})$. The most expensive operation in lines 8-14 is estimating the cost of rewritings with set B materialized; the time complexity of this step is kN_a. Hence, the complexity of the algorithm is $O(N_a^2 k^{\frac{5}{2}})$.

7 Experimental Results

In order to evaluate our algorithms we conducted several experiments. The real dataset used for all these experiments was a collection of more than 10,000 NSF proposal abstracts. We processed each document and extracted a collection of roughly 2.5 million terms and 100,000 sentences. We calculated the frequencies for individual terms and term association scores for pairs of terms and pruned the entries according to our discussion in Section 4.

7.1 Cost Model Savings

In this experiment, we do a baseline comparison between a local optimal plan and an 'average' plan, in terms of the difference in estimated costs. For 'average query plan, we estimate the cost for all query plans and compute the average cost. For our testing, we generated 4 sets of queries, with the number of terms per query fixed in each set, but varied from 2 to 5 between sets. For each set we generated 5000 queries with terms chosen randomly from our term collection. To keep the naturalness of the queries, the selection process used frequencies so that terms with higher frequencies appear more often in our generated queries. The probability of selecting term τ is therefore proportional to the frequency of τ and is given by

$P(t = \tau) = f(\tau)/\sum_{t \in \mathbf{C}} f(t)$, where $\mathbf{C}$ is the corpus of terms.

After generating the queries, we calculate the selectivity for each query. The selectivity of a query is the expected number of sentences that contain all the terms of the query, and is given by the product of selectivities of its terms or $sel(Q) = \prod_{t \in N_Q} sel(t)$, where $sel(t)$ is the selectivity of term t, which is the ratio of sentences that contain t and N_Q is the set of query terms. The other parameter we compute is the standard deviation of the selectivities of the terms, denoted by $std(t)$. For each query we build all possible plans, and estimate the cost for each plan using our cost models.

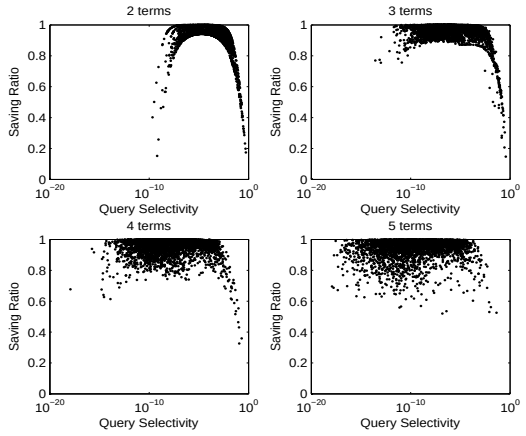

Figure 3. Saving Ratios vs. selectivities(log-scale)

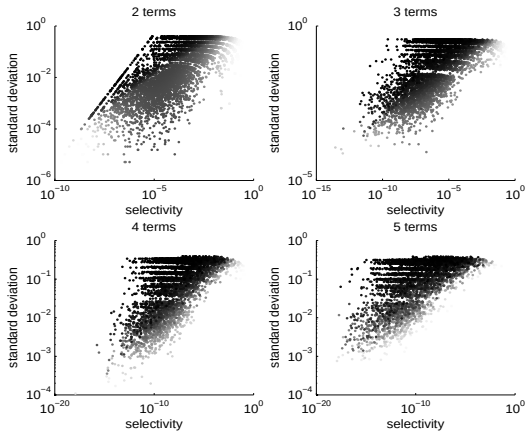

Figure 4. Spectrum of Saving Ratio vs. selectivity(log-scale) and standard deviation(log-scale)

The ratio of the saving, defined as (average cost - minimum local cost)/average cost, is shown for each set of our queries in Figure 3. As shown, the majority of the savings are close to 1, which indicates that in most cases a best plan has a much lower estimated cost. Also the saving is generally greater when the query selectivity is low, i.e. a small fraction of data is retrieved. This is expected because a high query selectivity is often the result of having only frequent terms in queries, and for such queries there is not much difference between the costs of a best plan and an average plan.

However, there are some exceptions; in particular, the plot of 2-term queries shows that not all queries with low selectivities benefit the most. If all query terms have low selectivities, it does not matter much which subset of the terms and in what order they are placed on a plan tree, and as a result there is not much difference between a best plan and an average plan in terms of the estimated number of I/Os. Figure 4 shows this intution for the same set of queries. The intensity of the darkness of each point gives a measure of the saving for a query, with darker points showing higher savings. The saving not only depends on query selectivity but also on the standard deviation of the individual term selectivities. For a fixed selectivity, the saving ratio increases as the standard deviation increases. The reason is that a higher standard deviation results in a greater difference between minimum local cost and the average cost, because best local plans can use low selectivity terms.

7.2 Savings over a Relational Database

To evaluate the accuracy of our cost model in a real setting and its applicability within a relational database framework, we tried to push our query plans to a commercial database engine and compared the costs (in terms of running time) to the costs of the plans fully generated by the relational engine. Most commercial databases that we are aware of impose restrictions that prevent one from passing a query plan. We used IBM DB2 as our relational DBMS, and generated a set of around 5700 random queries having 2 to 5 terms each. For each query we obtained two SQL queries: one query only had the terms of the best local plan and the other had all the terms of the query (referred to as a full filtering plan). Since DB2 does some kind of caching, different invocations of a query can result in different execution times. Therefore, we ran each query three times and only considered the minimum cost (cost of the query with the most caching). To make a fair comparison, we added the CPU overhead of our cost estimation to the execution times of best local plans on DB2. On a modest machine (PIII/933MHz with 2GB RAM), the overhead was on average 0.93ms, 1.8ms, 2.99ms and 4.36 ms for queries with 2, 3, 4 and 5 terms respectively.

Figure 5 shows the savings in the execution times of best local plans over the full filtering plans. The majority of the queries have a saving greater than or equal to zero, which

means that DB2 has a smaller running time for the best local plan we find over a plan that contains all query terms. Our experiment shows that on average, best local plans run approximately 1.8 times faster on DB2. For queries with more terms, the saving is higher on average. The average savings are 0.09, 0.20, 0.28 and 0.31 for queries with 2, 3, 4 and 5 terms, respectively. The amount of the saving is less than our estimated saving over average plans. There are two reasons for this: first, we could only pass our term selection but not ordering to DB2; second, DB2 was doing its own optimization on query expressions of both full filtering and best plans. That said, in order to show the statistical significance of the difference, we conducted a student's t-test with $t = 2.892$ and $n > 120$. the probability that our plans executed faster than DB2 plans is by chance is less than 0.0025.

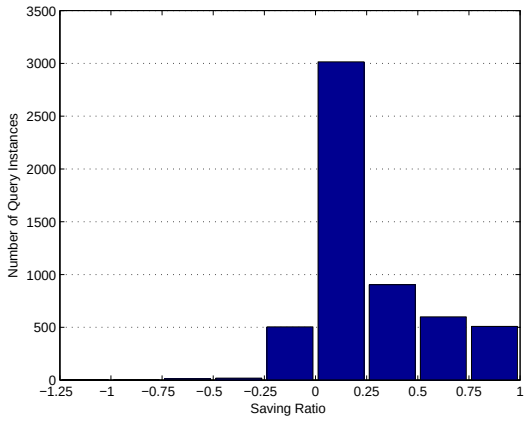

Figure 5. The histogram of the savings of db2 running times for 5568 queries

7.3 Adding Rewritings

In the presence of multiple rewritings, finding an optimal plan is computationally expensive, as shown analytically; hence we came up with a greedy algorithm which was significantly faster. The objective of our experiment in this section is to evaluate the effectiveness of our greedy algorithm. For our rewriting set, we generated a number of query seeds and used each seed to produce a set of rewritings by replacing, adding or removing terms from the query seed. We created 2 sets of query seeds, each with 100 seeds; query seeds in one set had 4 terms and in the other set had 5 terms. Query seeds were generated by selecting random terms from our collection of terms. For each query seed of size n, the rewritings were generated by randomly replacing, adding or removing r terms, where r varied from 1 to $n-1$, giving $n-1$ different sets of rewritings. The probability of replacing terms was 0.5, while probability of adding and removing terms were 0.25. All our term selections adhered to the probability given in Section 7.1 Finally, we ended up with 700 rewriting sets, each having 10 rewritings.

For each rewriting set, we find and estimate the costs of a sub-optimal plan using CSGreedy, an optimal plan using an exhaustive search, and a plan that consists of local optimal plans for each rewriting. We find the saving for both CSGreedy and optimal strategies over the sum of minimum local costs. Figure 6 compares the savings of CSGreedy and exhaustive optimal search for 429 rewriting sets that were discussed. Each data point on this figure gives the greedy saving of one set of rewritings. Since the horizontal axis shows the savings of optimal plans, any point on line *y=x* represents a query for which CSGreedy finds an optimal plan. All the data points must be under the diagonal line of *y=x* because no rewriting set can have a greedy saving greater than optimal. Also the closer the data point is to the optimal line, the better it estimates the optimal solution. As this figure shows, there are many rewriting sets for which CSGreedy finds an optimal solution. Moreover, there are only a few number of rewriting sets for which CSGreedy cannot find a solution better than the sum of minimum local costs. The amount of saving of CSGreedy is also very much compareable to that of optimal; on average the saving is within 92% of the saving of an optiaml plan.

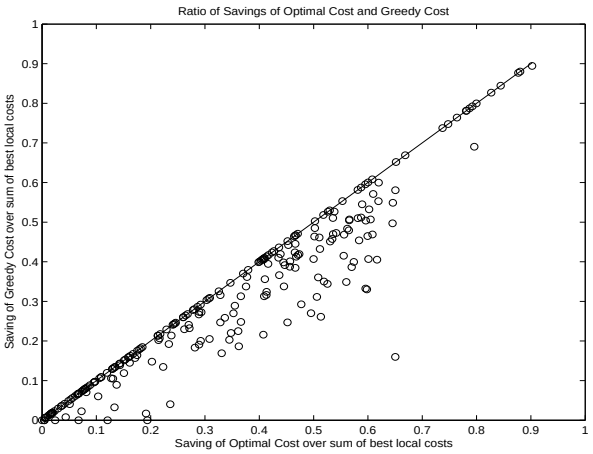

Figure 6. The Saving of Optimal Cost vs. the Saving of Greedy Cost

8 Conclusion and Future Extensions

In this paper, we propose using relational databases for querying natural language text and address the issues of mapping and optimizing queries. Our theoretical results show that we can significantly reduce the size of the search space for the plans of a NLTQ, while guaranteeing to find an optimal plan. We also show that finding the overall optimal plan for a NLTQ and its set of rewritings is NP-Hard

in general. Therefore, we propose an efficient greedy algorithm with expteced $O(N_a^2 k^{\frac{5}{2}})$ time complexity where N_a is the average number of terms per rewriting and k is the number of rewritings. Our experimental results show that the estimated costs of our local optimal plans are usually an order of magnitude less than the costs of average plans, and that query selectivity and standard deviation of term selectivities are two major factors that determine the amount of saving. The actual execution times for our queries mapped into SQL show that our cost model performs well in estimating the cost, and that taking into account the overhead for optimizing queries, it is beneficial to find the optimal plan on a commercial relational database. Our final results on optimizing a set of multiple rewritings demonstrate that our CSGreedy algorithm performs well compared to an optimal plan, with an average saving within 92% of the saving of an optiaml plan.

To the best of our knowledge, this is the first work that studies the issues related to querying and query optimization over natural language text in the context of relational databases. As a future extension we can consider the study of joining result set of NLTQ's with other relational data for different sizes of join. We would expect that our estimated optimal plans on average would outperform a random plan or a plan which chooses all the terms of the query. Moreover, as the size of the join increases we would expect to see a larger gap between the performance of our optimal plans and any random plan on average. As another future work, it would be interesting to somehow conduct experiments to compare the effect of optimizing multiple rewritings on a commercial relational database framework.

References

[1] Pubmed. `http://www.ncbi.nlm.nih.gov/entrez`.

[2] Kdd cup 2002, 2002. `http://www.biostat.wisc.edu/~craven/kddcup/`.

[3] Oracle text, an oracle technical white paper, 2005. `http://www.oracle.com/technology/products/text/pdf/10gR2text_twp_f.pdf`.

[4] Eugene Agichtein. *Extracting Relations From Large Text Collections*. PhD thesis, Columbia University, 2005.

[5] Dirk Bahle, Hugh E. Williams, and Justin Zobel. Efficient phrase querying with an auxiliary index. In *SIGIR*, pages 215–221, 2002.

[6] Sergey Brin. Extracting patterns and relations from the world wide web. In *WebDB*, pages 172–183, 1998.

[7] Surajit Chaudhuri, Umeshwar Dayal, and Tak W. Yan. Join queries with external text sources: Execution and optimization techniques. In *SIGMOD Conference*, pages 410–422, 1995.

[8] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE*, pages 227–238, 2004.

[9] Nancy Chinchor. Muc-7 named entity task definition. In *Seventh Message Understanding Conference (MUC-7)*, 1998.

[10] Mariano P. Consens and Tova Milo. Algebras for querying text regions: Expressive power and optimization. *Journal of Computer and System Sciences*, 57(3):272–288, 1998.

[11] Open Text Corporation. Enterprise content management solutions (ecm). `http://www.opentext.com`.

[12] Stefan Deßloch and Nelson Mendonça Mattos. Integrating sql databases with content-specific search engines. In *VLDB*, pages 528–537, 1997.

[13] Roy Goldman and Jennifer Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *SIGMOD Conference*, pages 285–296, 2000.

[14] Haobin Li and Davood Rafiei. Dewild: a tool for searching the web using wild cards. In *SIGIR*, page 731, 2006.

[15] Albert Maier and Hans-Joachim Novak. Db2's full-text search products - white paper, 2006.

[16] Marius Pasca, Dekang Lin, Jeffrey Bigham, Andrei Lifchits, and Alpa Jain. Names and similarities on the web: fact extraction in the fast lane. In *ACL*, pages 809–816, 2006.

[17] Prasan Roy, Srinivasan Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 249–260. ACM, 2000.

[18] Airi Salminen and Frank William Tompa. Pat expressions : an algebra for text search. In *COMPLEX*, 1992.

[19] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[20] Ellen M. Voorhees and Dawn M. Tice. Building a question answering test collection. In *SIGIR*, pages 200–207, 2000.

Appendix

Lemma 1. *The number of query plans for a NLTQ with N terms is given by $\sum_{n=1}^{N} qss(N,n)$ where*

$$qss(N,n) = \begin{cases} N & n = 1 \\ \frac{1}{2}\sum_{i=1}^{n-1} qss(N,i)qss(N-i,n-i) & n > 1. \end{cases}$$

Proof of Lemma 1. In order to find the search space size for plans with n nodes, selecting their terms from N different terms from the query, we first need to enumerate the different orientations that a plan tree can take with n

leaves and then we will find how many different permutations can N different terms make on such a tree. For example, the number of different orientations that a tree can take with $n = 1$ or $n = 2$ is only one. These are shown in Figure 7.(a). Therefore, the number of different trees with $n = 1$ is $\binom{N}{1} = N$ when we have N terms to distribute over leaves.

For $n \geq 2$ we can solve this problem recursively and solve for the left and right subtrees. Therefore, we can have k leaves in the left subtree and $n - k$ leaves in the right one. We denote such a tree with $(k)(n-k)$. However, it turns out that $(k)(n-k)$ and $(n-k)(k)$ give the same filtering sequence and have the same plan cost. This is shown for a query plan with $n = 3$ in Figure 7.(b). It turns out that for each plan there is a symmetric plan that does the same filtering and can be obtained by rotating the tree horizontally. Therefore, we have to halve the total number of query plans possible for a tree with n leaves. This results in the $qss(N, n)$ recursive function. In order to find the overall space size, we just need to sum over trees with different numbers of leaves, from 1 to N, which gives the formula for our lemma.

□

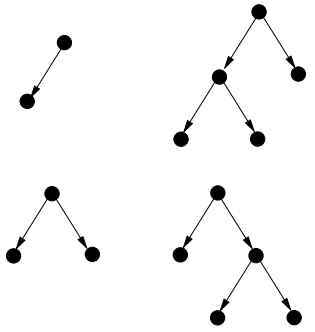

Figure 7. Examples of binary trees for different number of leaves

Theorem 1. *Assuming terms independence, for a query plan with n leaves represented in a Left Deep Tree (LDT), the lowest estimated cost can always be obtained by sorting terms according to their frequencies and placing lower frequency terms on leaves in higher depths.*

Proof of Theorem 1. Figure 8.(a) shows our target LDT. terms are sorted according to their frequencies and lower frequency terms are placed at the bottom of the tree. For such a tree, we have

- $\forall\, i, j \leq n, i \leq j \Rightarrow f_i \leq f_j$
- $\forall\, i, j \leq n, f_i \leq f_j \Rightarrow H(i) \geq H(j)$

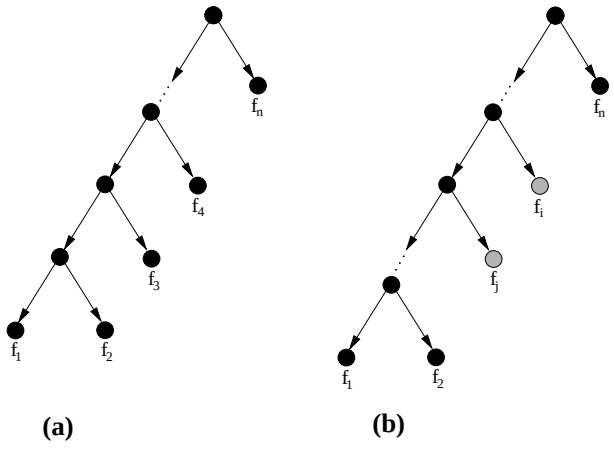

Figure 8. LDTs used for representing query plans. (a) Best query plan LDT with n leaves (b) terms on i_{th} and j_{th} leaves have been replaced

Where $H(k)$ is the height of node (leaf) k. Cost of this ordered LDT is given by

$$\begin{aligned} C_1 &= \min\left(C_a + f_1(\frac{C_t}{C_p} + C_a), 2C_a + \frac{C_t}{C_p}(f_1 + f_2)\right) \\ &+ \sum_{k=3}^{n} \min\left(f_{1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \end{aligned}$$

Where $f_{1..m}$ is the frequency of the resulting subtree and is given by $S_t \times \prod_{i=1}^{m} \frac{f_i}{S_t}$.

Figure 8.(b) is the same as the LDT to it's left except that terms i and j have been swapped. Therefore, in this LDT we have $i < j$ but $H(j) < H(i)$. The cost of this unordered LDT is given by

$$\begin{aligned} C_2 &= \min\left(C_a + f_1(\frac{C_t}{C_p} + C_a), 2C_a + \frac{C_t}{C_p}(f_1 + f_2)\right) \\ &+ \sum_{k=3}^{i-1} \min\left(f_{1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \\ &+ \min\left(f_{1..i-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_j\right) \\ &+ \sum_{k=i+1}^{j-1} \min\left(f_{1..i-1,j,i+1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \\ &+ \min\left(f_{1..i-1,j,i+1..j-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_i\right) \\ &+ \sum_{k=j+1}^{n} \min\left(f_{1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \end{aligned}$$

We can easily show that the following two relations hold

- $\forall\ a, b, c \in R,\ \ a \leq b \Leftrightarrow \min(a, c) \leq \min(b, c)$.
- $\forall\ \ s, s\prime \in P(\{1, \ldots, n\}),\ \ s\prime \subset s \Rightarrow f_{s\prime} \geq f_s$, where $P(s)$ gives the power set of s.

Comparing C_1 and C_2 shows that LDT costs are equal for the $i-1$ lower leaves and $n-j$ upper leaves. These are first, second and last statement of C_2 which are equal to their corresponding costs in C_1. Moreover, using the relations above we can easily show that the fourth statement of C_2 is always less than or equal to it's corresponding cost in C_1, which is

$$\begin{aligned} & \sum_{k=i+1}^{j-1} \min\left(f_{1..i-1,j,i+1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \\ \leq & \sum_{k=i+1}^{j-1} \min\left(f_{1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \end{aligned}$$

Therefore, we only need to compare the costs at leaves i and j. To complete the proof, we need to show that the following always holds

$$\begin{aligned} & \min\left(\underbrace{f_{1..i-1} \cdot C_a}_{A}, \underbrace{C_a + \frac{C_t}{C_p} \cdot f_j}_{B}\right) \\ + & \min\left(\underbrace{f_{1..i-1,j,i+1..j-1} \cdot C_a}_{D}, \underbrace{C_a + \frac{C_t}{C_p} \cdot f_i}_{C}\right) \\ \leq & \min\left(f_{1..i-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_i\right) \\ + & \min\left(\underbrace{f_{1..j-1} \cdot C_a}_{E}, C_a + \frac{C_t}{C_p} \cdot f_j\right) \end{aligned}$$

We can see that $A \geq D \geq E$ and $B \geq C$. We have

$$\min(A, B) \geq \min(A, C)$$

$$\begin{aligned} I. \min(D, C) = D & \Rightarrow E \leq D \leq C \leq B \\ & \Rightarrow \min(E, B) = E \\ & \Rightarrow C_2 \geq C_1 \\ II. \min(D, C) = C & \Rightarrow C \leq D \leq A \leq B \\ \min(D, C) + \min(A, B) & = \begin{cases} C + A & A \leq B \\ C + B & B \leq A \end{cases} \end{aligned}$$

$$\begin{aligned} A \leq B & \Rightarrow \min(A, C) + \min(E, B) = C + E \\ & , \quad E \leq A \Rightarrow C_2 \geq C_1 \\ B \leq A & \Rightarrow \min(A, C) + \min(E, B) = C + \min(B, E) \\ & , \quad \min(B, E) \leq B \Rightarrow C_2 \geq C_1 \end{aligned}$$

□

Theorem 2. *Assuming terms independence, a query plan t with n leaves has a traversal equivalent left deep tree that always has an estimated cost less than or equal to the estimated cost of t.*

Proof of Theorem 2. An LDT has the property that it only has one leaf pair at the lowest level of the tree and no other leaf has any leaf siblings. Any non-LDT has at least one extra leaf pair. we compare the cost of a leaf pair and cost of two leaves on consequent levels of a tree as will appear in an LDT. Figure 9.(a) shows an LDT and Figure 9.(b) shows one of it's traversal equivalent Bushy trees. As Theorem 1 suggests, we assume the frequencies of the LDT are sorted and assume we have the same frequencies for our traversal equivalent bushy tree. Using Figure 9 and cost formulas, we compute the cost of the LDT and bushy subtrees as follows.

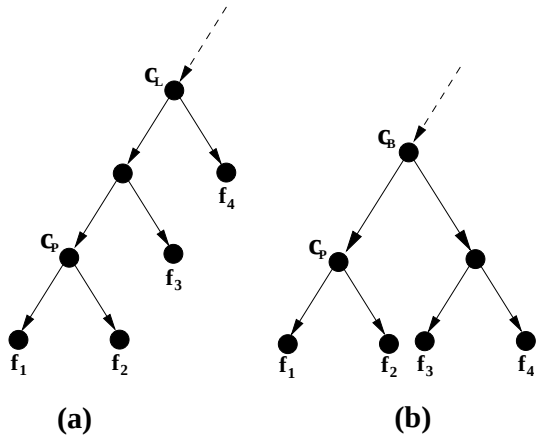

Figure 9. An LDT versus bushy tree. Bushy trees have at least one leaf pair more than LDTs

$$\begin{aligned} c_L = & \min\left(f_r C_a, C_a + \frac{C_t}{C_P} f_3\right) + \\ & \min\left(\frac{f_r f_3}{S_t} C_a, C_a + \frac{C_t}{C_p} f_4\right) + c_P \end{aligned} \quad (1)$$

$$\begin{aligned} c_B = & \min\left(C_a + f_3\left(\frac{C_t}{C_p} + C_a\right), 2C_a + \frac{C_t}{C_P}(f_3 + f_4)\right) \\ & + 2C_a + \frac{C_t}{C_p}\frac{f_3 f_4}{S_t} + \min\left(f_r C_a, C_a + \frac{C_t}{C_p}\frac{f_1 f_2}{S_t}\right) + c_P \end{aligned} \quad (2)$$

Where c_L is the cost of the LDT subtree and c_B is the cost of the bushy subtree. Moreover, we can easily show that

$$a + b \leq c \Rightarrow \min(a, x) + \min(b, y) \leq c \tag{3}$$

Where x and y could be any numbers. Using Equations(1),(2) and the above formula, we have

$$\left(C_a + \frac{C_t}{C_P} f_3\right) + \left(C_a + \frac{C_t}{C_p} f_4\right) \leq 2C_a + \frac{C_t}{C_P}(f_3 + f_4)$$

Using relation 3 and adding c_P to both sides of inequality, will result in the following inequality.

$$c_L \leq c_P + \underbrace{2C_a + \frac{C_t}{C_P}(f_3 + f_4)}_{m_1} \tag{4}$$

similarly, we have

$$\begin{aligned} &\left(C_a + \frac{C_t}{C_P} f_3\right) + (c f_r f_3 S_t C_a) \leq C_a + f_3 \left(\frac{C_t}{C_p} + C_a\right) \\ &\Rightarrow c_L \leq c_P + \underbrace{C_a + f_3 \left(\frac{C_t}{C_p} + C_a\right)}_{m_2} \end{aligned} \tag{5}$$

Finally, using inequalities (4),(5) we will have $c_L \leq c_P + \min(m_1, m_2)$ and this proves our theorem or $c_L \leq c_B$ □