Indexing Text Data under Space Constraints

Bijit Hore
University of California Irvine
bhore@ics.uci.edu

Hakan Hacigumus
IBM Almaden Research
hakan@acm.org

Bala Iyer
IBM Silicon Valley Lab
balaiyer@us.ibm.com

Sharad Mehrotra
University of California Irvine
sharad@ics.uci.edu

ABSTRACT

An important class of queries is the LIKE predicate in SQL. In the absence of an index, LIKE queries are subject to performance degradation. The notion of indexing on substrings (or q-grams) has been explored earlier without sufficient consideration of efficiency. q-grams are used to prune away rows that do not qualify for the query. The problem is to identify a finite number of grams subject to storage constraint that gives maximal pruning for a given query workload. Our contributions include: i) a formal problem definition, proof that the problem is NP-hard and adaptation of a previously studied approximate algorithm that produces results within a provable error bound, ii) performance evaluation of the application of the novel method to real data, and iii) parallelization of the algorithm, scaling considerations and a proposal to handle scaling issues.

Categories and Subject Descriptors: H.2.4[Systems]: Relational databases, Textual databases; H.3.1[Content Analysis and Indexing]: Indexing methods; H.2.2[Physical Design]: Access methods

General Terms: Algorithms, Experimentation

Keywords: Like queries, SQL, Index, B-tree, q-grams

1. INTRODUCTION

In this paper we study the problem of designing efficient indexing techniques to support SQL LIKE queries over string data. In SQL, through the LIKE clause, UNIX style, wild card queries can be specified. Two special characters '_' and '%' are used to specify "any single character match" and "any substring match" (including the empty string) respectively. LIKE queries are a subclass of textual pattern matching queries that have been extensively studied in the literature ([22], [23], [16], [18], [4]). [11] provides an excellent survey of various data structures/algorithms developed for pattern queries. Unfortunately, none of these specialized data structures are supported as an access method by modern DBMSs. Given the complexity of incorporating new data structures in DBMSs ([27, 28][1]), solutions employing existing access methods already supported by DBMSs - viz., B+trees [15], and bitmap indices [26] are more practical and hence more useful.

Motivated by the above mentioned practical considerations, we explore a **q-gram** based indexing approach in which strings are indexed based on a set of q-grams they contain. A q-gram is defined as follows:

Definition 1.1. **q-gram:** *For a given alphabet $\sum$, a q-gram is defined as any string of symbols from $\sum$ of length q. We will refer to a q-gram as a* **gram** *in general where its length can be an arbitrary positive integer (usually small).*

Now let us see how one can support SQL LIKE queries over a dataset (attribute) of strings using a gram based approach.

Evaluating LIKE queries using q-grams: First, a set of grams I are chosen to index a database of strings R. For each gram $g \in I$ there is a pointer to every string $s \in R$ that contains g. For a pattern P specified in the LIKE query, the set of all common grams, i.e. the set $I(P) = G(P) \cap I$ are extracted, where $G(P)$ is the set of all substrings in P. It is easy to see that any string containing P also contains each of the q-grams in $I(P)$. Therefore a (super)set of strings containing P is returned by retrieving all strings that contain some/all of the q-grams in $I(P)$. Lastly the false positives are pruned out to determine the exact set.

While the gram-based technique for evaluating LIKE queries is straightforward, it raises several non-trivial issues that require deeper analysis:

- How should *candidate* grams be generated and which of the candidate grams should be chosen to build the index? As should be obvious the choice of grams impacts the number of false positives that need to be pruned which impacts performance of queries.
- What data structure should be used to index the selected grams and how should the queries be processed given the data structure[2]?

[1] Incorporating data structures into DBMSs requires developing solutions to concurrency control, recovery, efficient updates, disk-based storage and retrieval etc .... But the problem with most of these data structures is, that they cannot be easily adapted to support these features. As a result database vendors have not integrated these into their systems.

[2] Given an index, multiple approaches to evaluate a query may exist since a query can be answered using any subset of grams it contains. In such a situation, the most efficient way of evaluating a query can be posed as an optimization problem

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CIKM'04, November 8–13,2004,Washington,DC,USA
Copyright 2004 ACM 1-58113-874-1/04/0011 ...$5.00.

In this paper we focus on the first of the above two challenges. First, we formally define the problem of gram selection. Specifically, given a set of grams G, database strings R, and queries Q, the gram selection problem consists of choosing a set of grams from G that minimize the number of false positives in evaluating Q over R. We show that even though the gram selection problem is NP-Hard, a constant factor approximation algorithm with polynomial complexity exists. While the approximation algorithm is of polynomial complexity, it does not scale to large data sets or a large number of queries. We, therefore, examine two optimization approaches in this context: i) **parallelization**: where we break the original problem into smaller sub-problems which can be solved in parallel. ii) **workload reduction**: where we reduce the size of workload Q by selectively retaining some of the queries from Q and discarding others. The weights of the retained queries are also suitably adjusted to reflect this change.

This paper makes the following contributions: **i)** To the best of our knowledge, this is the first work to formally define the gram selection problem. The paper shows that while the gram selection problem is NP-hard, constant factor approximation can be achieved in polynomial time. **ii)** Approaches to exploit parallelism and workload reduction are developed to scale the approach to large data and query sets. Techniques for gram selection have previously been explored in the literature [1, 6, 5, 7]. However, unlike our work, these approaches are somewhat ad-hoc. We discuss a few of these approaches in greater detail in the section 7.

The remaining sections of the paper are as follows: In section 2, we formalize the gram selection problem and show it to be NP-hard. In the same section we develop a constant-factor approximation to the gram selection problem. Parallelization and workload reduction methods are explored in sections 3 and 4 respectively. In section 5, we discuss a way of generating candidate grams. In section 6, experimental results of our gram selection algorithm for a real bibliographic database is reported and compared with a prototype implementation of the algorithm proposed in [1]. In section 7, we present a brief overview of related work. Conclusions are drawn in section 8.

2. GRAM SELECTION

We begin this section by formalizing the gram selection problem. We show it to be NP-Hard and adapt an approximation algorithm for a generic "Set Cover" problem to our case. Finally we analyze its time and space complexity.

2.1 Formalizing the Gram Selection Problem

Let $Q = \{q_1, \ldots, q_{n_Q}\}$ be a set of query strings (workload). $R = \{r_1, \ldots, r_{n_R}\}$ be the set of strings in the database and $G = \{g_1, \ldots, g_{n_G}\}$ be the set of candidate grams. As stated earlier, the gram selection problem is to choose a subset of grams from G such that the total number of false positives is minimized for queries in Q. To decide which set of grams to be included in the index in order to minimize the false positives, one needs to calculate the *benefit* of adding a gram to the index. The *benefit* of a gram g to a query q that contains it, is equal to the number of database strings it prunes from R (i.e. the number of records **not** containing g). Thus the total benefit of g, denoted by $benefit(g)$, is equal to the product of the number of records pruned by g and the number of queries that contain g as a substring (of course the benefit of a gram g to a query not containing it is 0). Given the benefits associated with each gram, the simple greedy algorithm to select the best k grams would be to choose the k grams with the highest values of benefit out of all candidate grams in G. But unfortunately, this greedy heuristic does not result in the optimal selection as can be seen from the following example:

Example 1: Consider an instance where $G = \{an, ra, ne\}$; $R = \{SanFrancisco, NewYork, Newark\}$ and workload $Q = \{\%San\%, \%Fran\%, \%kane\}$. The benefits associated with the grams are $benefit(g_1) = 2 * 3 = 6$, $benefit(g_2) = 2 * 1 = 2$ and $benefit(g_3) = 1 * 1 = 1$. If the above mentioned greedy algorithm is used to choose the top-2 grams, one would select an and ra in that order. But since all the records pruned by gram ra for query $\%fran\%$ (the only query is appears in) are already pruned by gram an (which also appears in the same query), there is no additional benefit due to gram ra. Instead choosing ne is more beneficial, as it prunes the record *San Francisco* for the query $\%kane$, which is not pruned by an. ♢

The above example illustrates that *incremental benefit* and not the absolute benefit is more useful for selecting an optimal set of grams. In this case, the order in which grams are added to the index also become important. Therefore selection of an optimal set of grams is not straight-forward and requires deeper analysis.

This problem can also be visualized in a graph-theoretic setting: Consider the graph **H** with the vertex-set $V = G \cup R \cup Q$. Let there be an edge (g, q) between a vertex $g \in G$ and $q \in Q$ iff q **contains** g as a substring (denoted as $g \in q$). Also an edge (g, r) exists between a vertex $g \in G$ and $r \in R$ iff r **does not contain** g as a substring (denoted as $g \notin r$). We say that the gram g **enables** the pair (q, r) where $q \in Q$ and $r \in R$ iff there exists a path of length 2 between q and r with g as the intermediate node (we will use the terms "covers" and "enables" interchangeably from here on). Therefore an optimal set of grams would maximize the total number of **distinct** (q, r) pairs covered by its grams. The example below illustrates the graph visualization and is used as the running example in the remainder of the paper.

Example 2: Let the database R contain the records shown in table 1, the query workload (assumed to be only on the second attribute) be $Q = \{$ "_an%"; "%York%"; "San Franc%"; "%kane%"$\}$ and the candidate set of grams (given in advance) for the index be $G = \{$"an", "or", "ne"$\}$.

CODE	AIRPORT
SJC	San Jose, California
LAX	Los Angeles International
JFK	John F Kennedy, New York
LGA	La Guardia, New York
EWR	Newark
SFO	San Francisco
OAK	Oakland

Table 1: Airports and Codes

The associated graph **H** is shown in figure 1. For instance the gram $g2$ enables the pairs $(q2, r1)$, $(q2, r2)$, $(q2, r5)$, $(q2, r6)$ and $(q2, r7)$ as seen from the graph.♢

Given that the gram selection problem can be visualized as a graph, the simplest version of problem, where the goal is to select the best k grams, can be formally stated as follows:

Definition 2.1. **Top-k-Grams(G,R,Q,k):** *For a given k find a $G' \subseteq G$ of size k, such that number of distinct (q, r) pairs covered by grams in G', is maximized over all possible subsets of size k.*

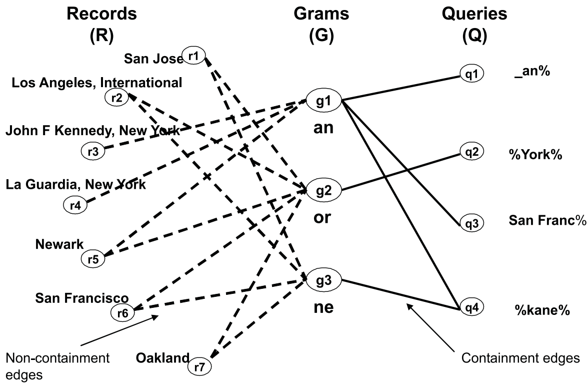


Figure 1: Graph Visualization

The above definition implicitly assumes that each gram has an *unit cost* and each query has an *unit weight*. But in the general case the *cost*[3] associated with each gram and the *weight*[4] of queries may vary (can be arbitrary positive numbers). In addition one may specify the maximum available space as a *budget constraint* on the index-size. Now assume that besides the sets R, Q and G, the following parameters and functions are also specified:

1. A weight function for queries: $weight(q) : Q \rightarrow \Re$ denoting the importance/frequency of the query string q ($\Re$ stands for the real numbers).
2. The cost function $cost(g) : G \rightarrow \Re$
3. The budget constraint **M**.
4. A one-to-many mapping for the candidate set of grams: $cover(g) : G \rightarrow Q \times R$, the set of all (q, r) pairs covered by g (this can be explicitly computed from the sets G, R and Q as seen in example 2). The following extensions are also made:
   - for a set of grams $I \subseteq G$, we define $cover(I) = \bigcup_{g \in I}(cover(g))$ and $weight(I) = \sum_{q_i \in Q}(weight(q_i) \times |cover_{q_i}(I)|)$ where $cover_{q_i}(I)$ denotes all (q, r) pairs enabled by $g \in I$ and $q = q_i$.

Definition 2.2. *Given the above, an instance of the gram selection problem (in the general case), is to select an optimal subset of grams, denoted by* **BestIndex**(Q, R, G, M)*, such that:*

$$\textbf{BestIndex}(Q, R, G, M) = I_{max} \subseteq G \text{ such that}$$

$$weight(I_{max}) \text{ is maximum over all } I, \text{ where } I \subseteq G \text{ and}$$

$$cost(I_{max}) = \sum_{g \in I_{max}} (cost(g)) \leq M$$

The BestIndex problem can be shown to be NP-Hard by a straight forward reduction from the *Set Cover* problem which is known to be NP-Hard [13]. Note that Top-k-Grams, is a special case of the BestIndex problem where each gram has an *unit cost*. It can be shown that Top-k-Grams problem is NP-Hard as well. (Refer to [8] for proof.)

[3] *cost* may vary with the indexing mechanism employed, eg. for B+tree the cost of a gram g is the number of leaf level pointers that need to be maintained for g, which is equal to the number of records containing g

[4] *weight* may denote importance or frequency of a query pattern

2.2 A Constant Factor Approximation Algorithm for BestIndex

The BestIndex problem defined above can be shown to form subclass of a more general class of problems called the *Budgeted Maximum Coverage of Sets* (BMC) problem. We refer the interested reader to appendix A for a brief description of the BMC problem and how we can represent an instance of the BestIndex class as an instance of BMC problem.

Khuller [3] presents a $\frac{1}{2}(1 - \frac{1}{e})$ factor approximation algorithm for the BMC. It guarantees a set-cover of weight $\frac{1}{2}(1 - \frac{1}{e})w(OPT)$ where $w(OPT)$ is the weight of the optimum cover that is possible within some specified budget L (if weight of each element is 1 then $w(OPT)$ simply denotes the number of elements in the optimum cover). We adapt this algorithm to the gram selection problem and present a similar $\frac{1}{2}(1 - \frac{1}{e})$ factor approximation algorithm for BestIndex in figure 3.

Before presenting the basic algorithm for gram selection, two measures need to be defined: *benefit* and *utility* of a gram g. In general benefit is defined with respect to current set of grams in the index I and is denoted by $benefit(g, I)$ or simply $benefit(g)$ when it is clear from context.

Definition 2.3. *The quantities $benefit$ and utility for a candidate gram g are defined as follows:*

1. **benefit(g, I)** *or* **benefit(g)** *= Total weight of* **new** *(q, r) pairs covered by g that are not already covered by some $g' \in I$*
2. **utility(g)** $= \frac{benefit(g)}{cost(g)}$ *(i.e. benefit of g per unit cost)*

For instance if a candidate gram g covers $\{(q_1, r_1), (q_2, r_2), (q_3, r_3)\}$, but the pair (q_1, r_1) is already covered by some other gram in the index, then $benefit(g) = 2$ instead of 3.

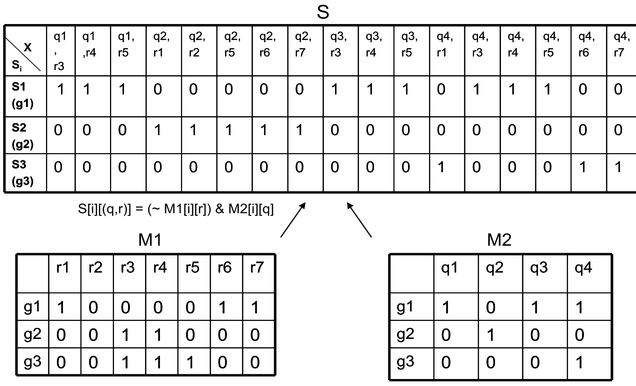
The **BestIndex-Naive** algorithm is given in figure 3 which is adaptation of the greedy approximation algorithm of [3]. The algorithm needs to store information regarding $cover(g)$ for all grams. In effect it needs to generate all possible (q_i, r_j) pairs that belong to some $cover(g)$. This is denoted by the matrix S in figure 2. But since S could be very large in practice, we generate it on the fly using two smaller matrices $M1$ and $M2$ instead. The example below illustrates the process.

Example 3: Consider the problem of example 2. We store the two matrices $M1$ and $M2$. $M1$ is the *gram-record containment* matrix and $M2$ is the *gram-query containment* matrix. $M1$ has a 1 in the slot $[i][j]$ iff $g_i \in r_j$ otherwise has a 0, similarly for $M2$. The diagram of figure 2 illustrates how to generate the matrix S using a couple of simple logical operations to combine the entries from the two smaller matrices $M1$ and $M2$. (Note that unlike in the graph model, in our implementation we choose to denote the *containment relationship* between grams and records in the matrix $M1$ (i.e. 1 if a gram is present in a record else 0). This helps in making the matrix sparse leading to smaller subproblems when partitioned as seen in section 3.) $\diamondsuit$

The BestIndex-Naive algorithm use $M1$ and $M2$ to generate S on the fly but we omit that step from figure 3.

2.3 Complexity of BestIndex-Naive Algorithm

The algorithm **BestIndex-Naive** needs to iteratively select the next best candidate gram to be added to the already selected set of grams (index) till either all elements ((q, r) pairs) that can be covered by some gram in G, are actually covered or the allocated budget is used up. In each iteration,



X / S_i	q1, r3	q1, r4	q1, r5	q2, r1	q2, r2	q2, r5	q2, r6	q2, r7	q3, r3	q3, r4	q3, r5	q4, r1	q4, r3	q4, r4	q4, r5	q4, r6	q4, r7
S1 (g1)	1	1	1	0	0	0	0	0	1	1	1	0	1	1	1	0	0
S2 (g2)	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
S3 (g3)	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	1

	r1	r2	r3	r4	r5	r6	r7
g1	1	0	0	0	0	1	1
g2	0	0	1	1	0	0	0
g3	0	0	1	1	1	0	0

	q1	q2	q3	q4
g1	1	0	1	1
g2	0	1	0	0
g3	0	0	0	1

Figure 2: Set theoretic representation

Algorithm 2: BestIndex-Naive(Q, R, G, M)
while some (q, r) uncovered AND space available
  for every gram $g \in G \setminus I$ set $benefit[g] = 0$
  for every query $q_k \in Q$
    for every record $r_j \in R$
      if (pair (q_k, r_j) not covered by a $g \in I$) *//verification*
        for every candidate $g_i \in G \setminus I$
          if (pair (q_k, r_j) covered by g_i) **then**
            $benefit[g_i] = benefit[g_i] + 1$
  if ($\exists$ a g with $benefit[g] > 0$) **then**
    for every candidate g
      $utility[g] = \frac{benefit[g]}{cost(g)}$
  else EXIT
  $I = I \cup \{g_{max}\}$, where g_{max} has maximum utility
end

Figure 3: The basic greedy gram selection algorithm

BestIndex-Naive needs to compute the incremental benefit of each remaining candidate gram. Therefore the time complexity is given by:

THEOREM 1. *Worst case time-complexity of BestIndex-Naive to compute an index with n grams is* $\mathbf{O}(\mathbf{n}\times \mid \mathbf{R} \mid \times \mid \mathbf{Q} \mid \times \mid \mathbf{G} \mid)$ *if the verification step could execute in constant time.*

Proof: The term $\mid R \mid \times \mid Q \mid$ originates from the fact that those many possible elements could be present in the universal set that need to be covered. Since there is no explicit storage, we generate the elements by computing the Cartesian product of R and Q (figure 2 illustrates the procedure). In our implementation the verification step is proportional to size of index I at that stage. But since for most cases $\mid I \mid$ is small compared to $\mid G \mid$, we can approximate the time taken by a constant.

THEOREM 2. *Worst case space complexity of BestIndex-Naive is* $\mathbf{O}(\mid \mathbf{G} \mid \times(\mid \mathbf{Q} \mid + \mid \mathbf{R} \mid))$

Proof: The space required is mainly due to the two boolean matrices $M1$ ($\mid G \mid \times \mid R \mid$) and $M2$ ($\mid G \mid \times \mid Q \mid$).

$M1$ stores a *true* for a (g, r) pair if g is a substring of r and *false* otherwise. Similarly $M2$ stores a *true* bit in the (g, q) cell if g is a substring of q. (Note that a pair (q, r) is covered by a gram g iff $M1[g, r] = false$ and $M2[g, q] = true$).

For large size of Q, R and G, the running time of the algorithm can become prohibitively large, as can be seen in table 2 in section 6. Therefore we investigate optimization techniques for scaling the naive algorithm to larger input sets. In the next section we describe parallelization of the BestIndex algorithm.

Algorithm 3: BestIndex-Improved(Q, R, G, M)
Q-G-list is a vector indexed by q where each slot
  Q-G-list$[q]$ is set of $g's$ s.t. g substring of q
G-R-list is a vector indexed by g where each slot
  G-R-list$[g]$ is set of $r's$ s.t. g is a substring of r
$R_c = \{r \in R \mid \exists g \in G$ where g is a substring of $r\}$
  (R_c computed using the Q-G-list and G-R-list)
while some (q, r) uncovered AND space available
  for every gram $g \in G \setminus I$ set $benefit[g] = 0$
  for every query $q_k \in Q$
    for every record $r_j \in R_c$
      for each gram $g \in$ Q-G-list$[q_k] \setminus I$
        if (g is not a substring of r_j AND
          (q_k, r_j) not covered by any $g \in I$) **then**
            $benefit[g] = benefit[g] + 1$
  if($\exists g$ with $benefit[g] > 0$) **then**
    for every candidate g
      $utility[g] = \frac{benefit[g]}{cost(g)}$
  else EXIT
  $I = I \cup \{g_{max}\}$, where g_{max} has maximum utility
end

Figure 4: The improved greedy BestIndex algorithm

3. PARALLELIZABLE ALGORITHM

The BestIndex-Naive algorithm described in section 2 has poor scale up properties, both in terms of space and time as noted in section 2.3. We make several optimizations to make it much faster in practice and scale well with problem size both in terms of time and space requirements. The following three paragraphs summarize the principal optimization that were carried out :-

1) Pruning: In the preprocessing phase we prune out some very *frequent* grams from the candidate set G. For instance we pruned out all grams that had a selectivity of more than 0.1 in most cases (i.e. grams that are present in 10% or more of the records). The threshold selectivity was chosen based on some runs of the algorithm on small workloads using the complete set of candidate grams. We noted the maximum selectivity of the grams chosen and decided the cutoff point on that basis. The quality of the index is not affected much by this pruning, which is quite evident from results of our experiments. At the same time, one must point out that most of the grams are *infrequent* (have selectivity below threshold) and therefore continue to remain in the candidate set during the actual run of the experiments. Adding all the infrequent grams to the index is also not an option as their combined weight is much larger than the allocated budget, therefore optimization is required. From performance perspective pruning helps in reducing the size of the subproblems generated as we see below.

2) Auxiliary data structures: *Q-G-list* and *G-R-list*, these two adjacency lists store *id* of all grams in a query q in the slot Q-G-list$[q]$ and *id* of all records containing gram g in G-R-list$[g]$ respectively. The two auxiliary data structures can be populated initially in the pre-processing phase. This makes the innermost loop of the BestIndex-Naive algorithm (figure 3) proportional to a small constant c_1 rather than $\mid G \mid$, where c_1 is the average number of grams in a query string. The resulting algorithm, **BestIndex-Improved** is shown in figure 4.

3) Partitioning and Parallelizing: Here we exploit the sparseness of the boolean matrices $M1$ and $M2$. One must note that these two matrices are generally sparse in most cases when selectivity of each candidate gram is small and hence a lot of compaction is possible. Our approach

Algorithm 4: Parallelizable-BestIndex(Q, R, G, M, k)
Partition Q into k disjoint sets $Q_1, \ldots, Q_k$
Create corresponding sets $R_i \subseteq R, G_i \subseteq G, i = 1 \ldots k$
where $G_i = \{g \mid g \in G \wedge g$ substring of any $q \in Q_i\}$
$R_i = \{r \mid r \in R \wedge \exists g \in G_i$ s.t. g is a substring of $r\}$
while some (q, r) uncovered AND space left
  for every gram $g \in G \setminus I$ set $benefit_{global}[g] = 0$
  over all subproblems $i = 1, \ldots, k$
    compute $benefit_i[g], \forall g \in G_i$
    $benefit_{global}[g] = benefit_{global}[g] + benefit_i[g]$
  if ($\exists g$ with $benefit[g] > 0$) **then**
    for all $g \in G \setminus I$
      $utility[g] = \frac{benefit_{global}[g]}{cost[g]}$
  else EXIT
  $I = I \cup \{g_{max}\}$ where g_{max} has maximum *utility*
  Update information in subproblems containing g_{max}
end

Figure 5: The Parallelizable BestIndex algorithm

is the following: We cluster queries into small groups using the following distance measure between two queries q_1 and q_2: $Dist(q_1, q_2) = \frac{|(G_1 - G_2) \cup (G_2 - G_1)|}{|G_1 \cap G_2|}$ where G_1 and G_2 denote the set of candidate grams occurring in q_1 and q_2 respectively. In practice this results in a good partition on the workload Q which leads to small subproblem sizes. Subproblem generation is nothing but a natural breakup of the original problem into smaller domains where they can be solved more or less independently as shown in the **Parallelizable-BestIndex** algorithm in figure 5. We profit from the fact that both space and time requirements of each of the (say) k subproblems generated, falls by a factor $c_2 k (c_2 > 1)$ on an average, as compared to the BestIndex-Improved algorithm (of figure 4) working on the complete problem. Hence the resources required to solve the union of all the subproblems is lesser than what is needed for the original problem. We found this factor c_2 to be approximately 5 or 6 for most inputs. To illustrate, let the original problem take some T amount of resource (time + space), then if we break it into 10 subproblems, empirically we found each of the 10 subproblems take about $\frac{T}{(c_2 * 10)}$ amount of resources. Therefore solving each of the 10 subproblems independently (almost), we note a gain of $c_2 (\approx 5$ in our experiments) in terms of resources. Here we should point out what is not shown explicitly in the parallel algorithm of figure 5, that it uses the BestIndex-Improved algorithm of figure 4 to compute the benefit of grams in each subproblem.

The new Parallelizable-BestIndex algorithm takes an input parameter k which denotes the number of subproblems to be generated. We choose k to be rather large ($O(\mid Q \mid)$). Ideally gain is maximized when $k = \mid Q \mid$. But in practice the pre-processing and book-keeping time increases with the number of subproblems when run on a single machine, therefore in our experiments we clustered a small number of "similar" queries into a single subproblem. We used a value of k between $\frac{|Q|}{5}$ and $\frac{|Q|}{2}$. We end the section with another small example illustrating the benefits of partitioning and parallelization approach.

Example 4: Continuing with the same example of airport database, we illustrate how the BestIndex problem can be solved more efficiently using our partitioning strategy. We have four queries in our database, $q_1, \ldots, q_4$, therefore if we decide to partition the problem into 4 sub-problems then the inputs to each subproblem are as shown in the figure 6 (refer to figures 1 and 2 in previous sections to

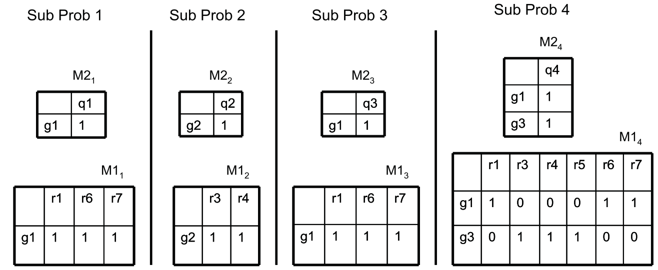


Figure 6: Sub-Problems resulting from partition

recall how to generate the inputs). Note that for the ith subproblem if the query set is Q_i, then to construct the matrix $M2_i$, the candidate set G_i contains only those grams which appear in some $q \in Q_i$. Similarly the set of records indexed in the matrix $M1_i$ need be only those which contain some $g \in G_i$. Therefore in our example, on an average the time taken to complete one iteration of the algorithm (i.e. to choose the next best gram) would require about $1*1*3+1*1*2+1*1*3+2*1*6 = 20$ logical operations between element of $M1_i$ and $M2_i$ (over all subproblems i), whereas without partitioning, it would have been proportional to $3*4*7 = 84$ logical operations as one can see from the matrices $M1$ and $M2$ in figure 2.$\diamondsuit$

The algorithm of figure 5 is highly parallelizable since an iteration can be carried out in each subproblem independently from others. One only needs to make a small global comparison per iteration (between k local best candidates) to choose the next overall best candidate gram to be added to the index. Time did not permit us to carry out implementation of the algorithm using parallel hardware, but even the speedup obtained on a sequential machine was substantial.

4. WORKLOAD REDUCTION

In this section, we describe another approach that we explored, to address the scaling problem by **reducing the size of the workload**. We experimented with a distance-based clustering approach to compress the workload and propose a few interesting distance measures.

As outlined in section 3, the running time of the BestIndex-Improved and Parallelizable-BestIndex algorithm were made literally independent of the size of the candidate gram-set by maintaining auxiliary data structures (albeit at the cost of a little extra preprocessing). But still in the worst case scenario a factor of ($\mid Q \mid \times \mid R \mid$) is unavoidable depending on the nature of queries and grams generated. This will cause the running-time of even the Parallelizable-BestIndex algorithm to increase rapidly as $\mid R \mid$ and $\mid Q \mid$ increase. We decided not to alter the set of records and instead explored possibilities of reducing the size of the input query workload Q. The main goal is to minimize the *degradation of quality* while reducing the workload by discarding some queries. The notion of quality degradation is explained in the following paragraphs.

Quality of workload reduction is dependent on how well the *relevant characteristics* (relevant to the application) of the original workload Q are preserved in the reduced workload Q'. Clustering based SQL workload compression for index-tuning application has been studied in [12]. They took a distance based clustering approach in that work and the key is to design the appropriate *distance* measure suitable for the application at hand. The main step is to choose the right representative for a group of pruned queries and

fold them onto this representative query, which is retained. The notion is that, *better* the representation of the pruned queries in the retained set (i.e. the reduced workload) *nearer* will be the performance of the application to what it would be on the original (uncompressed) workload, which is the desirable goal. In our case, the application is of course selection of best grams for index construction.

The authors in [12] deal with reducing (compressing) a generic SQL query workload where the distance between SQL queries is based upon the query signatures (the set of attributes the query accesses) and selectivity of each attribute in the queries. The goal there was to get the index-selection tool to *perform well* using just the compressed workload. That is, to suggest a good set of indices to build on the data which benefits the original set of SQL queries. Of course this makes the tacit assumption that working with the uncompressed dataset would result in the selection of most beneficial indexes for the dataset to start with (which seems to be the common sense). We take a similar approach to workload reduction as outlined below :-

1. Define a suitable computable distance measure **dist**() on the query space (this distance measure need not be a metric).
2. Decide a compression factor t. That is after reduction we want $k = \lfloor \frac{|Q|}{t} \rfloor$ queries to be retained. (We chose t based on the maximum size of the workload that can be handled within the available memory of the system).
3. Carry out the k-median clustering algorithm on Q. When the clusters have stabilized, *fold* all the queries in a cluster onto the median query of that cluster. Folding of queries in a cluster consists of pruning all remaining queries other than the median query and adding the weights of these pruned queries to that of the median query. These k cluster-medians that are retained, form our reduced workload Q'. In effect these cluster medians are the representatives of the set of pruned queries from the respective clusters.

4.1 Maximum Deviation Distance

The key step to achieve good reduction is determined by how well the distance measure captures the notion of *loss in quality* when index is built using a reduced workload Q' instead of the original Q. We measure quality of an index I, for a query set Q by the statistic *aggregate-proportion-of-error(Q,I)* also denoted by $APE(Q,I)$ or simply APE when there is no confusion. APE is defined as:

$$APE(Q,I) = \frac{\sum_{q \in Q} \text{\# false strings returned for q using I}}{\sum_{q \in Q} \text{\# true strings containing q}}$$

Therefore the quality of an index I' built using Q' with respect to the original query set Q is determined by the statistic *aggregate-proportion-of-error(Q,I')* (i.e. $APE(Q,I')$). The objective of our gram selection algorithm is to minimize the quantity $APE(Q,I')$.

Now we propose the **MaxDevDist** family of distance functions based on APE and experimentally compare its performance to random sampling and edit-distance based reduction (in section 6). The $MaxDevDist$ measure is based on the following assumption:

MaxDevDist Assumption: *For a given Q, R, G and budget M, BestIndex-Naive algorithm (and its variants) construct the index I_{Best} that minimizes the value of $APE(Q,I)$ over all indexes I that can be created in a "respectable" polynomial time using the information Q, R, G and M.*

In other words the assumption states that the BestIndex algorithm with input Q, R, G and M will construct an index that prunes out the maximum total number of false positives (for queries in Q) over all "respectable" polynomial-time algorithms (by respectable, we mean practical.). Any other index would only result in a higher APE. Let us call this the *ideal-index* and the index computed using Q' as the *reduced-index*. Therefore any *reduced-index* would have a higher APE than the *ideal-index*.

$MaxDevDist(q_1, q_2)$ is computed as if q_1 needs to be folded onto q_2 and in general the measures need not be a metric. To simplify computation we consider effect of just the set of grams present in these two queries q_1 and q_2 in isolation from all other grams in the candidate set. Folding causes nodes of kind (q_1, r) to collapse into nodes of type (q_2, r) and transferring of weights to the latter. This causes erroneous computation of $benefit(g)$ for some grams $g \in G$ which results from two factors:-

1. Loss of some *benefit-enhancing* (q, r) pairs for g.
2. Increment in weights of some queries due to folding.

By the assumption stated above, this would lead to a loss of quality and hence result in a higher APE (i.e. $APE(Q,I') \geq APE(Q, I_{Best})$). We tried to combine these effects into a single distance measure and came up with a few variations. First we see an example of the folding process and then the formal definitions follow.

Example 5: Continuing with the airport example, we have $\mid Q \mid = 4$. Recall that the 4 queries were q_1 = _an%, q_2 = %York%, q_3 =San Franc% and q_4 = %kane%. Let the weights of the queries be $w_1 = 1; w_2 = 3; w_3 = 2; w_4 = 1$ respectively. Now if we want to reduce the workload to a size of 2, we make 2 clusters. Let the clusters formed be $\{q_1, q_3, q_4\}$ and $\{q_2\}$ with the median queries as q_1 and q_2 respectively. The reduced workload will therefore be $Q' = \{q_1, q_2\}$ with weights $w'_1 = 1 + 2 + 1 = 4$ and $w'_2 = 3$. ♢

$\mathbf{MaxDevDist(q_1, q_2)}$ class of distance measure assigns a numeric distance between two queries q_1 and q_2 when the former is being folded onto the latter. Since $benefit(g)$ for a gram is dependent on the number of record it prunes we propose that the distance measure be **directly proportional to sum of the maximum deviation in** $benefit(g)$ of each gram $g \in (G_1 - G_2) \cup (G_2 - G_1)$ where G_i is set of all grams occurring in q_i. Some variants of the measure also take into account the similarity between queries and therefore are **inversely proportional** to the **similarity** between q_1 and q_2 as well.

We suggest the following variants of the generic distance measure. (Note: In the definitions given below, $\overline{R(g)}$ denotes the set of records that do **not** contain the gram g).

1. $\mathbf{MaxDevDist_1(q_1, q_2)} = \frac{\sum_{g_n} |\overline{R(g_n)}|}{\sum_{g_d} |\overline{R(g_d)}|}$
   where $g_n \in (G_1 - G_2) \cup (G_2 - G_1)$ and $g_d \in (G_1 \cap G_2)$
2. $\mathbf{MaxDevDist_2(q_1, q_2)} = \frac{\sum_{g_n} |\overline{R(g_n)}|}{\sum_{g_d} |\overline{R(g_d)}|}$
   where $g_n \in (G_1 - G_2) \cup (G_2 - G_1)$ and $g_d \in G_2$
3. $\mathbf{MaxDevDist_3(q_1, q_2)} = \sum_{g_n} \mid \overline{R(g_n)} \mid$, where $g_n \in (G_1 - G_2) \cup (G_2 - G_1)$.

Example 6: Let there be two query patterns : $q_1 = ab$ and $q_2 = bb$. Set of candidate grams for q_1 is $G_1 = \{a, b, ab\}$

and for q_2 is $G_2 = \{b, bb\}$, therefore $G_1 - G_2 = \{a, ab\}$; $G_2 - G_1 = \{bb\}$ and $G_1 \cap G_2 = \{b\}$. Let $| R |= 10$. Let cost associated with the grams a, b, ab and bb be 7, 5, 2 and 1 respectively (i.e. the number of records that contain these grams). Also if weight associated with q_1 be $w_1 = 1$, folding q_1 onto q_2 results in increment of w_2 by 1 unit. Now the various distance measures between q_1 and q_2 are:
$MaxDevDist_1(q_1, q_2) = \frac{|\overline{R(a)}|+|\overline{R(ab)}|+|\overline{R(bb)}|}{|\overline{R(b)}|} = \frac{3+8+9}{5} = 4.0$
where $\overline{R(g)}$ is the set of records pruned by g. Similarly for the second one
$MaxDevDist_2(q_1, q_2) = \frac{|\overline{R(a)}|+|\overline{R(ab)}|+|\overline{R(bb)}|}{|\overline{R(b)}|+|\overline{R(bb)}|} = \frac{3+8+9}{5+1} \approx 3.67$ and
$MaxDevDist(q_1, q_2) = | \overline{R(a)} | + | \overline{R(ab)} | + | \overline{R(bb)} | = 3 + 8 + 9 = 20.0$ ♢

5. CANDIDATE SET GENERATION

So far we have assumed that candidate set of grams has been available to us all the while. There are many ways in which G may be generated, from the set of records R, set of query patterns Q or from both the sets R and Q (possibly). Authors in [1] generate their candidate set from R. We generated the set of candidate grams from the given set of queries Q, which represent the patterns of interest to us. We describe our method below, in detail. Let us define a set G to be a **Perfect Discriminating Set** with respect to set Q and R (i.e. $G = PDS(Q, R)$) if by evaluating any $q \in Q$ using G, we retrieve the **exact** set of records from R that satisfy q (i.e. no false-positives are retrieved). It is obvious that the set of all sub-strings appearing in any query string in Q forms a $PDS(Q, R)$. Therefore one can achieve zero relative error for any $q \in Q$ by inserting each of its constituent strings as a key in the index. For example in the query string "San Franc%" we would insert the string "San Franc" as a single gram in the index. Similarly for "Oak%California", two grams would be inserted,"Oak" and "California" and so on. But in trying to do so, two problems arise:

Size of Index: If Q is large, then a PDS such as one mentioned above, will be very large as well. Therefore we cannot store all of these in the index, given the space restrictions. As a result one needs to generate a more compact set of candidate grams that are common to many query strings.

Over-fitting: Choosing short, common grams from Q for the candidate set is preferable as we do not want to over-fit the index to patterns in Q. This will result in poor performance of queries which are not represented well in the query workload. Since any finite query set or query model cannot capture the space of all possible queries perfectly, over-fitting is likely to degrade performance[5]. Avoiding over-fitting is even more desirable when we consider workload-reduction using compression techniques. As we saw in section 4 a reduced workload uses only a *representative* subset of queries and therefore the performance of the index on the actual set of queries might degrade if it has been over-fitted to the reduced set.

Candidate Set generation: Below we outline how to generate the candidate set of grams using a **suffix tree**:

1. Build a suffix tree using all $q \in Q$ (we use an implementation of Ukkonen's algorithm [4]).

[5]This is in a way analogous to the desire of fitting a smooth curve to a data set instead of piece-wise linear curve though the latter will reduce the error metric in many cases (for example least-squares metric)

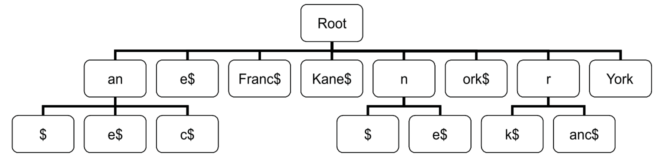


Figure 7: Suffix Tree : Example

2. Generate the set of all *path-labels* from the root to each node of the suffix tree, call it G_0. Path label of a suffix-tree node is simply the concatenation of each *edge-label* from the root to that node.

3. Construct the candidate set of grams G from G_0 by just retaining the **shortest prefix** of each string in G_0 so that they have a selectivity below some predecided threshold c and still remain *distinguishable* from each other. We call these grams *prefix-path-labels*. (The term *path-labels* would imply the complete path label from the root to a node). We illustrate the procedure by the following example.

Example 7: For generating the *prefix-path-label*, assume that a sequence of nodes from root to a leaf with their *edge-labels* in the suffix tree are the following: *(n1, "Fran"), (n2, "cis"), (n3, "co")*. Then corresponding to the nodes n1, n2 and n3, we will generate the prefix-path-labels *"Fra"*, *"Franc"* and *"Francisc"*[6]. The point to note is that though "Fra" has selectivity less than the required threshold c, for n2 we generate a prefix that is *distinguishable* from the one generated for n1, hence minimally extend the edge-label "Fran" to "Franc". This is required since the string "Fran" is probably present in more number of queries than "Franc"and due to the nature of our algorithm, these two strings might have completely different *benefits* (and/or *utilities*) associated with them (refer to previous section for definition of the terms). ♢

Example 8: Continuing with example 2 of section 1, we have the same dataset with two attributes *(Code, Airport)* and an index to be built on the *Airport* attribute. The query workload Q = { _an%, %York%, San Franc%, %kane%}. That is queries of type: "SELECT Code, Airport FROM R WHERE text LIKE *pattern*". Here *pattern* could be any of the four strings from Q. Figure 7 displays the suffix tree that is built from these four strings. In the figure each node displays the *edge-label* of the edge joining it to its parent node. The set of path labels that will be generated would be G_0 = { *an, anc, ane, e, Franc, kane, n, ne, ork, r, rk, ranc, York* }. But a post-processing might lead to discarding the grams from G_0 which have a *selectivity* greater than some threshold c $(0 < c < 1)$ say like "e", "n" and "r". Also we would add only the **smallest prefix** of the remaining grams from G_0 to G which have selectivity just less than c and are *distinguishable* from each other. That might result in adding just the strings *"Fra"* and *"Y"* to final candidate set G instead of the *Franc* and *York* and similarly for the other grams in G_0 as well. ♢

As an observation, in our experiments we found that a typical index of size 2MB discards more than 80% of the candidates in many cases. For example, in one of the runs on an input workload of 4,000 queries, the selected set contains about 2,800 out of the total 14,700 candidate grams.

In comparison to our methodology, the authors in [1] choose the indexing grams in three different ways : i) the

[6]assuming both F and Fr have selectivity $> c$

complete set of grams up to length 10; ii) all grams of length up to 10 that have selectivity below a threshold c, they call it the *multigram* set and iii) a set including only the **smallest common suffixes** of the *multigram* set that have a selectivity below c, they refer to it as the *shortest suffix set.* Of course their gram generation takes only the database into consideration and use no explicit query model/workload.

6. EXPERIMENTAL EVALUATION

In this section we summarize the experiments that we carried out and discuss their results. The dataset on which all our experiments were done is the real bibliographic data available from the Digital Bibliography & Library Project [2]. The schema consists of two character attributes, `Author`, that stores author name(s) delimited by comma if there is more than one author, and `Publication`, that stores the name of the publication. In rest of the section 1) we briefly describe the experimental setup and the various datasets. 2) Present the performance of the indexes constructed using the Parallelizable-BestIndex algorithm, on various query workloads. We also compare our performance with a similar algorithm to what was presented in [1] and implemented in the FREE system (Fast Regular Expression Indexing Engine). 3) We present experiments testing *resilience* of an index on workloads that incrementally *deviate* from the original workload and compare it with performance of the FREE algorithm. 4) Finally we present some results using reduction techniques discussed in section 4.

6.1 Setup and Performance Metrics

Database (R) consisted of 305,798 tuples each having two attributes (Author-Names, Title-of-Publication). For query workload Q we varied the size from 1000 to 4000 in steps of 1000. In the absence of a real query-workload, we generated a synthetic workload consisting of randomly chosen author last names from the *bag* of all author names occurring in the DBLP dataset (here a *bag* refers to a collection where multiple occurrences of an element are allowed). The candidate gram set (G) for a given set Q of queries were simply the *prefix-path-labels* of the suffix tree built on the strings in Q as described in section 5.

All experiments were carried out on a Dell workstation with a 2G-Hz Intel-IV processor, 512MB memory and 80GB hard-disk. Below we define the two main measures used for testing query performance on the various indexes constructed (the latter has already been defined in section 4).

Average Relative Error(Q,I) (ARE): the average fraction of false-positives retrieved in the solution set of a query for a given query workload Q and a given index I. This is our main measure of **quality** of an index. *Average Relative Error* (ARE) is defined as

$$\frac{1}{|Q|}\sum_{q\in Q}\frac{\text{\# false positives retrieved for q}}{\text{Total \# of records retrieved for q}}$$

Aggregate Proportion of Error (APE): for a workload Q is the ratio of total number of false positive returned to the total number of correct records returned, put mathematically, it is

$$\frac{\sum_{q\in Q}\text{\# false positives retrieved for q}}{\sum_{q\in Q}\text{\# true records for q}}$$

$\mid Q \mid = 100, \mid R \mid = 305798,$	$\mid G \mid = 500$	
Algorithm	**# iterations**	**(secs)**
BestIndex-Naive	1.52899×10^{12}	(1530)
BestIndex-Improved	9.17394×10^{10}	(100)
Parallelizable-BestIndex	3.0×10^{9}	(22)

Table 2: Running time of algorithms

6.2 Running Time of Algorithms

We summarize performance of the three algorithms in table 2 using a small input problem. We make the assumption that each query string on an average contains $c = 30$ grams (it is usually smaller than this) and a query on an average is present in not more than 5000 records at the most (this also is a loose upper bound). Further if the processor is able to perform 10^9 iterations per second, the number of iterations executed by each algorithm for choosing the top-100 grams is also shown in the table.

The factor of reduction in execution time while going from BestIndex-Naive to Parallelizable-BestIndex, is not necessarily as large as 500 as seen in the "# iterations" column. This is nominally offset by the increased pre-processing time for the parallelizable algorithm as compared to BestIndex-Naive. None the less for this small input problem, we saw a factor of 70 or more improvement in the running time in practice. Parallelizable-BestIndex algorithm runs roughly 5 times faster than BestIndex-Improved algorithm as reported in table 2.

6.3 Quality of Index

We discuss two sets of experiments here: *Quality of Index* and *Resilience of Index*.

Quality: We determine the quality of the index by evaluating query workloads of increasing size and computing the two statistics mentioned above. The smaller the values of the *average relative error* and *aggregate proportion of error*, better the index. We compared our performance to the FREE algorithm [1] which is also a gram based indexing algorithm, but meant for regular expression matching on text databases. We implemented a simplified version of the same which we refer to as the FREE algorithm from here on. The setting of the problem is a little different there as they do not impose any space constraint on the index. Also [1] does not consider any query-model or workload with respect to which the index needs to be optimized. Their underlying dataset consists of web pages unlike ours, where the dataset of interest is a particular column of a relational database table. But since regular expressions semantics subsume SQL-LIKE query semantics, the two algorithms are definitely comparable. To impose space restriction on the FREE algorithm, we terminate its gram-selection as soon as the index-weight reaches the allocated budget. The DBLP dataset used, was about 30MB of data. We allowed the index size to grow to a maximum of 2, 4 and 8 MB (which are roughly 6.67%, 13.33% and 26.67% of the complete database size). Assuming a B+tree as the index of choice, the number of total leaf pointers allowed for the 3 indexes were therefore 512K, 1024K and 2048K respectively. We make the assumption that each pointer takes 4 bytes and the size of the B+tree is approximately equal to the space taken by its leaf nodes. Figure 8 shows the performance of the 3 indexes on each of the 4 query workloads of sizes 1000, 2000, 3000 and 4000. In each of the experimental runs, the cutoff selectivity for both FREE and Parallelizable-BestIndex algorithm were kept equal to some c ($0.05 \leq c \leq 0.1$).

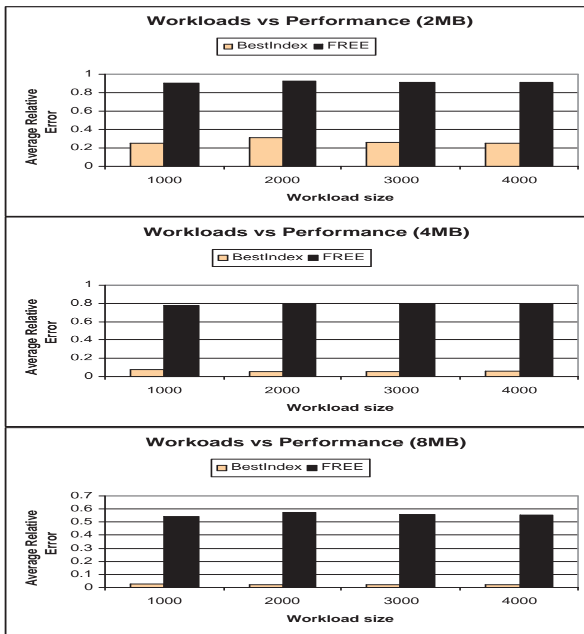


Figure 8: FREE vs BestIndex (Performance on various workloads)

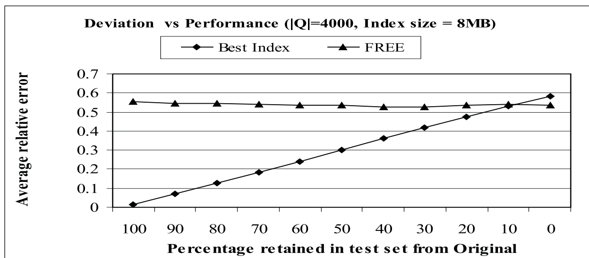


Figure 9: Performance under Deviation

Resilience: We also test how *resilient* our indexes are by letting the test query set deviate from the original workload used to build the index. The deviation of the test set is carried out at follows: We assume that we are given a *bag* of all possible queries, the universal set (bag). The original workload Q is considered to be a sample of size $\mid Q \mid$ from this universe. We create the "deviant" **test query workload** by retaining a fraction $f\times \mid Q \mid$ of the original queries (where $f < 1$) and randomly sampling the remaining $(1-f)\times \mid Q \mid$ queries from the universal set. Size of the workload was kept constant at 4000. Figure 9 plots the $Average\ Relative\ Error(ARE)$ values for both FREE and Parallelizable-BestIndex algorithms as f is decreased from 1 through 0 in steps of 0.1. As expected the performance of FREE was more or less constant, since it took no workload (or query model) into consideration. Performance of the BestIndex algorithm degraded more or less linearly with deviation, but as is evident from the graph, its performance is sufficiently better than FREE for the most part.

6.4 Workload Reduction

We carried out experiments using reduced workloads to evaluate the effectiveness of the various distance functions. For our experiments, the original workload Q contained 1000 queries. We tested with 3 different compression factors:- The workloads were reduced to 20%, 40% and 60% percent of the

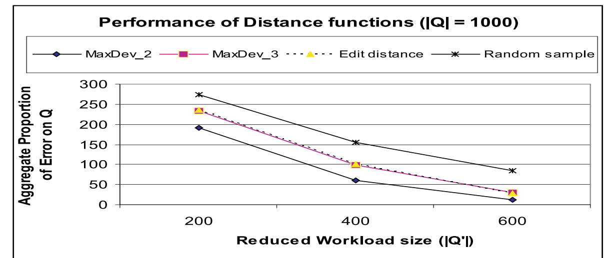


Figure 10: Performance of Distance Functions

original size and then used to build the index. We ran the k-median clustering algorithm using the different distance measures discussed in section 4. We also compared the performance of our distance functions with *random sampling* and *edit-distance* based compressions. Figure 10 shows the performance of the various reduced workloads. The plot shows two of the $MaxDevDist$ family of functions along with the random sampling and edit distance functions. We found $MaxDevDist_2$ to perform consistently better than the rest of the distance functions for all sizes of workloads. We present just one set of results due to lack of space. These results are only to be considered as indicative of the potential of the "distance-based clustering" method to compress workloads meaningfully as well as the effectiveness of the suggested distance measures.

7. RELATED WORK

The area of string indexing, searching, string-pattern matching (both for exact and approximate matching) has been researched extensively for decades. Needless to say there is a huge amount of work that is related to the context of the problem we address in this paper. The B+tree [15] and its variants, the SB-tree [16], prefix-B-tree [14] etc ... can be used to index strings and answer pattern matching queries. But there is no work that tries to optimize such indices under space constraints. In the following paragraphs we briefly mention some related work in areas of exact, approximate and regular expression string matching in textual or otherwise string databases (eg. DNA sequences).

Automaton based matching: This is perhaps one of the oldest approaches to string matching. There exists a vast amount of work in the area, [17], [19], [20]. Baeza-Yates [18] uses Patricia trees to model a text index and gives search algorithms that are logarithmic expected time in the size of the text for a class of regular expression. For dictionary searches a well known algorithm is the Aho-Corasick algorithm [21] which is also the basis of the *fgrep* command in UNIX. A thorough survey of the area is beyond the scope of this paper but we do believe that the main problem with most automaton techniques is because they are not optimized for secondary memory and hence for large data set their I/O performance degrades rapidly.

Secondary memory data structures: Exact matching can be carried out using Suffix trees [22] [4] and suffix arrays [23] amongst others, which are versatile data structures that are used to store all suffixes of a large string. They both take up linear space and have near optimal search complexity and optimal space complexity for external memory. Inverted lists [25] are popular data structures for indexing large text databases supporting complete word queries. Another secondary memory data structure is the SB-tree [16], it maintains a lexicographic order of all suffixes of a text un-

der *delete* and *insert* operations. The SB-tree has optimal I/O complexity for searches, inserts and deletes and optimal space complexity for storage. A good survey of approximate string matching literature can be found in [11]. But again none of these data structures really solve the problem we take on in this paper as they do not incorporate any notion of space constraint on the index they construct.

q-gram based filtering techniques: This set of prior work is nearest to ours in spirit. An important property used to design algorithms for q-gram based searches is that two strings which differ by a small edit-distance k will have many q-grams in common. [6] and [5] store an auxiliary table for *positional q-grams* of a given strings which needs to be queried and present SQL commands that extract all k-approximate strings to the query pattern. They extend the approach to compute approximate string joins and approximate substring matches. They have to maintain all q-grams of a given size for each database string thus perhaps requiring up to $O(n^2)$ extra space per attribute that needs to be queried. Another piece of similar work is [7], they apply the techniques to match patterns in DNA sequences by partitioning the base sequence into blocks and computing the number of common q-grams in these blocks and the query pattern of interest. The ones that have more than the threshold number of common q-grams are further scrutinized for approximate match. The closest piece of work to ours, that of [1] has been discussed extensively in the previous sections therefore we skip it here.

8. CONCLUSION

In this paper, we show that the gram selection problem is equivalent to the NP-Hard Budgeted Maximum Set Coverage Problem. This enabled us to exploit and adapt already available theoretical results and make significant advances over prior work. From a practical perspective, a) motivated by the observation that it is query performance that is the user's primary concern, we incorporated a query model into the gram selection problem, b) we introduced storage as an explicit constraint. As a result of our work, the gram selection problem can be solved within provable error bounds. Time and space complexity were the most important hurdle, which we have successfully overcome to a great extent by developing a scalable implementation of the basic greedy algorithm. We also explored quality preserving workload reduction techniques for scaling to larger workloads and suggested useful distance functions for clustering. By applying our results to a real database, we found our results superior to prior work that was neither query-cognizant nor storage-cognizant. The results are not just of theoretical significance but can have significant practical impact.

Acknowledgement: This work was funded by NSF grant number IIS-0220069

9. REFERENCES

[1] J. Cho and S. Rajagopalan. A Fast Regular Expression Indexing Engine. In *Proc. of ICDE*, 2002.

[2] Digital Bibliography & Library Project. http://dblp.uni-trier.de/.

[3] Khuller,S., Moss, A., and Naor, J. The Budgeted Maximum Coverage Problem. *IPL*, V 70, Num 1: 39–45, 1999.

[4] E. Ukkonen. Online construction of Suffix-trees. *Algorithmica*, 1993.

[5] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, L. Pietarinen, and D. Srivastava. Using q-grams in a dbms for approximate string processing. *IEEE Data Engineering Bulletin*, 24(4):28–34, 2001.

[6] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491-500, 2001.

[7] Burkhardt, S., Crauser, A., Ferragina, P., Lenhof, H., Rivals, E., Vingron, M. q-gram based Database Searching Using Suffix Array (QUASAR) RECOMB, 1999, pp. 77-83.

[8] Hore, B., Hacigumus, H., Iyer, B., Mehrotra, S. Indexing Text Data under Space Constraints *TR-DB-04-02*, www-db.ics.uci.edu/pages/publications/index.shtml

[9] D. S. Hochbaum. Approximating covering and packing problems: Set cover, vertex cover, independent set, and related problems. *Approximation algorithms for NP-hard problems, PWS Publishing Co., Boston*, 1996.

[10] D. S. Hochbaum and A. Pathria. Analysis of the Greedy Approach in Problems of Maximum k-Coverage. *Naval Research Quarterly*, (45):615-627, 1998.

[11] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31-88, 2001.

[12] S. Chaudhury. Compressing SQL Workloads. *ACM SIGMOD 2002*

[13] Garey, M. R, Johnson, D. S. Computers and Intractability: A Guide to Theory of NP-Completeness. Freeman, San Francisco, 1979.

[14] Bayer, R., and Unteraurer, K. Prefix B-trees ACM Trans. Database System. 2(1977), pp 11-26.

[15] Bayer, R., and McCreight, C. Organization and maintenance of large ordered indexes Acta Informatica, 1972, pp173-189.

[16] Ferragina, P., and Grossi, R. A fully-dynamic data structure for external substring search. ACM STOC, 1995, pp 693-702.

[17] Hopcroft, J., E., and Ullman, D. Introduction to automata theory, languages and computation. Addison-Wesley, 1979.

[18] Baeza-Yates, R., and Gonnet, G, H. Fast text searching for regular expressions or automaton searching on Tries. JACM, Vol 43, 1996, pp. 915-936.

[19] Crochemore, M., Hancart, C. Automata for Matching Patterns, Handbook of formal languages. Rosenberg, C., and Salaama, A. eds 2, Springer-Verlag, 1997, pp. 399-462

[20] Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M., T., and Seiferas, J. The smallest automaton recognizing the subwords of a text. Theoretical computer science, 40(1), 1985, pp. 31-55.

[21] Aho, A, V., and Corasick, M, J.. Efficient String matching : an aid to bibliographic search. Comm. ACM. 1975, pp. 332-340.

[22] McCreight, E., M. A space efficient suffix-tree construction algorithm. J. ACM 23, 1976, pp. 262-272.

[23] Manber, U., and Myers, G. Suffix Arrays a new method for on-line string searches. Siam *Journal on Computing 22*, 1993, pp. 935-948.

[24] Knuth, D., E. The Art of Computer Programming. Addison-Wesley, 1973 Vol 3: Sorting and Searching.

[25] Salton, G. Automatic Text Processing. Addison-Wesley, 1989.

[26] Chan, C., Y, and Ioannidis, Y., E. Bitmap Index Desing and Evaluation, ACM SIGMOD 1998. pp 355-366.

[27] Gray, J., Reuters, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Pub, 1993.

[28] Chakrabarti, K., Mehrotra, S. Efficient Concurrency Control in Multidimensional Access methods. SIGMOD, 1999, pp 25-36.

APPENDIX

A. THE BMC PROBLEM

The *Budgeted Maximum Coverage*(BMC) problem is defined in [3] as follows: A collection of sets $S = \{S_1, S_2, \dots, S_m\}$ with associated costs $\{c_i\}_{i=1}^m$ is defined over a domain of elements $X = \{x_1, x_2, \dots, x_n\}$ with associated weights $\{w_i\}_{i=1}^n$. The goal is to find a collection of sets $S' \subseteq S$ such that the total cost of elements in S' does not exceed a given budget L, and the total weight of the elements covered by S' is maximized. Again it is easy to show that the BMC problem is NP-Hard. [10] provides a simple $(1 - \frac{1}{e})$ greedy approximation algorithm for the special case where cost of each $S_i = 1$. The BestIndex problem can be reduced to an instance of the BMC problem as illustrated in the example below:

Example 9: Consider the problem of example 2, construct the set of elements $X = \{(q_i, r_j) \mid \exists g \in G \text{ s.t } g \in q_i \text{ and } g \notin r_j\}$ For each g_i ("an", "or" and "ne"), we create the corresponding subsets of X (i.e. consisting of exactly the pairs *enabled* by g_i), call these sets S_i. Assign weights to each $(q, r) \in X$ as $weight((q, r)) = weight(q)$ (1 here). Define $cost(g) = \mid \{r \mid r \in R \text{ and } g \in r\} \mid$. The various S_i's are represented in the adjacency matrix S shown in figure 2. $\diamondsuit$