# Answering Approximate String Queries on Large Data Sets Using External Memory

Alexander Behm[1], Chen Li[2], Michael J. Carey[3]

*Deptarment of Computer Science, University of California, Irvine*

{[1] abehm, [2] chenli, [3] mjcarey}@ics.uci.edu

*Abstract*— **An approximate string query is to find from a collection of strings those that are similar to a given query string. Answering such queries is important in many applications such as data cleaning and record linkage, where errors could occur in queries as well as the data. Many existing algorithms have focused on in-memory indexes. In this paper we investigate how to efficiently answer such queries in a disk-based setting, by systematically studying the effects of storing data and indexes on disk. We devise a novel physical layout for an inverted index to answer queries and we study how to construct it with limited buffer space. To answer queries, we develop a cost-based, adaptive algorithm that balances the I/O costs of retrieving candidate matches and accessing inverted lists. Experiments on large, real datasets verify that simply adapting existing algorithms to a disk-based setting does not work well and that our new techniques answer queries efficiently. Further, our solutions significantly outperform a recent tree-based index, BED-tree.**

## I. Introduction

Many information systems need to answer approximate string queries. Given a collection of data strings such as person names, publication titles, or addresses, an approximate string query finds all data strings similar to a given query string. For example, in data cleaning we want to discover and eliminate inconsistent, incorrect or duplicate entries of a real-world entity such as `Wall-Mart` versus `Wal-Mart` or `Schwarzenegger` versus `Schwartseneger`. Similar problems exist in record linkage where we intend to combine records of the same entity from different sources.

Various measures can quantify the similarity between two strings, such as edit distance, Jaccard, etc. Many approaches to answering approximate string queries rely on $q$-grams. The $q$-grams of a string are all its substrings of length $q$. For example, the 2-grams of string `cathey` are $\{\texttt{ca}, \texttt{at}, \texttt{th}, \texttt{he}, \texttt{ey}\}$. Intuitively, if two strings are similar, then they must share a certain number of grams. We can facilitate finding all data strings that share a gram by using an inverted index on the grams. For instance, Figure 1 shows a collection of strings and the corresponding inverted lists of their 2-grams. To construct the index, we decompose each data string into its grams and add its string id to the inverted lists of its grams.

To answer an approximate string query, we decompose the query string into its grams and retrieve their corresponding inverted lists. We find all the string ids that occur at least a certain number of times on the lists (see Section II). For those string ids we retrieve their corresponding strings, compute their real similarities to the query, and remove false positives. For many similarity measures, given a threshold on the similarity, a lower bound on the number of common grams between two similar strings can be computed [15].

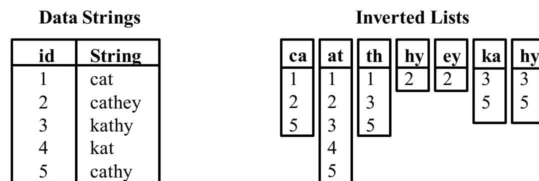



Fig. 1. A string collection of person names and their inverted lists of 2-grams.

**Motivation:** In this paper, we study how to answer approximate string queries when data and indexes reside on disk. Many existing works on approximate string queries have assumed memory resident data and indexes, but their sizes could be very large. Indexing large amounts of data may cause problems for applications. On the one hand, the indexes and data could be so large that even compression cannot shrink them to fit into main memory. On the other hand, even if they fit, dedicating a large portion of memory to them may be unacceptable. Database systems need to deal with many different indexes and concurrent queries, leading to heavy resource contention. In another scenario, consider a desktop-search application supporting similarity queries. It is infeasible to keep the entire index in memory when desktop search plays a secondary role in the users' activities, and hence we should consider secondary-storage solutions.

In recent years, major database systems such as Oracle [1], DB2 [2], and SQL Server [8] have each added support for some kind of approximate queries. Though their implementation details are undisclosed, they clearly must employ disk-based indexes. Our work could help to improve the performance of similarity queries in these systems.

**Contributions:** Storing the data strings and the inverted lists on disk dramatically changes the costs of answering approximate string queries. This setting presents to us new tradeoffs and allows novel optimizations. One solution is to simply map the inverted lists to disk, and use existing in-memory solutions for answering approximate string queries, retrieving inverted lists from disk as necessary. Such disk-based inverted indexes have been extensively studied for conjunctive keyword queries [22]. However, there are several key differences between our problem and the traditional problem of conjunctive keyword queries. First, a keyword query often consists of only a few words, and the opportunity of pruning results is limited to a few inverted lists. In contrast, an

approximate string query could have many grams, and it may be inefficient to "blindly" read *all* their inverted lists. Second, for conjunctive keyword queries an answer should typically appear on all the inverted lists of the query's keywords. In contrast, for an approximate string query, an answer must not necessarily occur on all the query grams' inverted lists. Third, keyword queries ask for possibly large documents, whereas approximate string queries ask for strings relatively smaller than documents. This difference in the size of answers makes it more attractive in our setting to ignore some inverted lists and pay a higher cost in the final verification step.

We make the following contributions. In Section III we propose a new storage layout for the inverted index and show how to efficiently construct it with limited buffer space. The new layout uses the unique filtering property of our setting (see Section II) to split the inverted list of a gram into smaller sublists that still remain contiguous on disk. We also discuss the placement of the data strings on disk to improve query performance. Intuitively, we want to order the data strings to reflect the access patterns of queries during post-processing. In Section IV we develop a cost-based, adaptive algorithm for answering queries. It effectively balances the I/O costs of accessing inverted lists and candidate answers. It becomes technically interesting how to properly quantify the tradeoff between accessing inverted lists and candidate answers when combining our new partitioned inverted index with the adaptive algorithm. Finally, Section V presents a series of experiments on large, real datasets to study the merit of our techniques. We show that the index-construction procedure is efficient, and that the adaptive algorithm considerably improves query performance. In addition, we show that our techniques outperform a recent tree-based index, BED-tree [21], by orders of magnitude.

### A. Related Work

In the literature, the term *approximate string query* also means the problem of finding within a long text string those substrings that are similar to a given query pattern. See [17] for an excellent survey. Here, we use this term to refer to the problem of finding from a collection of strings those similar to a given query string.

There are two main families of algorithms for answering approximate string queries: those based on trees (e.g., suffix trees) [16], [12], [21], and those based on inverted indexes on $q$-grams [11], [15], [5]. Suffix trees focus on substring-similarity search and mainly support edit distance. In this paper we focus on the inverted-index approach because it supports a variety of important similarity measures and it generalizes to other kinds of set-similarity queries. In Section V we compare our solutions to a recent tree-based approach, BED-tree [21], which is a disk-based index similar to a B-tree. Its main idea is to employ a smart global ordering of the data strings in the tree, and then transform a similariy query into a range query using the global ordering.

Several recent papers have focused on approximate *selection* queries [11], [15], [5] using in-memory indexes. Many algorithms have been developed for approximate string joins based on various similarity functions [3], [4], [9], [10], [18], [20]. Some of them are proposed in the context of relational databases. A recent paper [13] uses $q$-gram inverted indexes to answer substring queries. In this paper, we focus on approximate string selection queries, but some of our ideas may also apply to substring queries and approximate joins.

There are many studies on disk-based inverted indexes, mostly for information retrieval [22]. Index-construction procedures are presented in [22], and index-maintenance strategies are evaluated in [14]. Please refer to our introduction for a comparison between our problem and keyword queries.

Compression can improve transfer speeds of disk-resident structures. There are various compression algorithms tailored to inverted lists [23], [7]. These methods are orthogonal to the techniques described in this paper.

## II. Preliminaries

**Q-Grams:** Let $\Sigma$ be an alphabet. For a string $s$ of the characters in $\Sigma$, we use "$|s|$" to denote the length of $s$. The $q$-grams of a string $s$ are all its substrings of length $q$, obtained by sliding a window of length $q$ over $s$. For instance, the 2-grams of a string `cathey` are $\{\texttt{ca}, \texttt{at}, \texttt{th}, \texttt{he}, \texttt{ey}\}$. We denote the multiset of $q$-grams of $s$ by $G(s)$. The number of $q$-grams in $G(s)$ is $|G(s)| = |s| - q + 1$.

**Approximate String Queries:** An approximate string query consist of a string $r$ and a similarity threshold $k$. Given a set of strings $S$, the query asks for all strings in $S$ whose similarity to $r$ is no less than $k$. We also call such queries approximate *range* or *selection* queries. Various measures can quantify the similarity between two strings, as follows. The *edit distance* between two strings $r$ and $s$ is the minimum number of single-character insertions, deletions, and substitutions needed to transform $r$ to $s$. For example, the edit distance between "`cathey`" and "`kathy`" is 2. Other measures quantify the similarity between two strings based on the similarity of their $q$-gram multisets, for example:

- $Jaccard(G(r), G(s)) = \frac{|G(r) \cap G(s)|}{|G(r) \cup G(s)|}$
- $Dice(G(r), G(s)) = \frac{2|G(r) \cap G(s)|}{|G(r)| + |G(s)|}$.

**T-Occurrence Problem:** For answering approximate string queries we focus on $q$-gram inverted indexes. For each gram $g$ in the strings in $S$, we have a list $l_g$ of the ids of the strings that include this gram $g$, as shown in Figure 1 for 2-grams. An approximate string query can be approached by solving a "$T$-occurrence problem" [18] defined as follows: Find the string ids that appear at least $T$ times on the inverted lists of grams in the query string. $T$ is a lower bound on the number of grams any answer must share with $r$, computed as follows for edit distance. A string $r$ that is within edit distance $k$ of another string $s$ must share at least the following number of grams with $s$ [19]: $T_{ed} = (|r| - q + 1) - k \times q$. Intuitively, every edit operation can affect at most $q$ grams of a string, and $T$ is the minimum number of grams that "survive" $k$ edit operations. Similar observations exist for other similarity functions such as Jaccard and Dice [15]. Solving the $T$-occurrence problem

yields a superset of the answers to a particular query. To remove false positives we compute their real similarities to the query in a post-processing step. Our goal in this paper is to efficiently answer approximate string queries when both the data and indexes need to reside on disk.

**Generalization:** Set-based similarity measures such as Jaccard and Dice are not limited to $q$-grams. For example, we could also tokenize our strings into multisets of words (instead of grams), and use Jaccard or Dice to find similar multisets of words. In general, a *set-similarity query* is to find, from a collection of sets, those sets that are similar to a given query set. In this paper, we mostly use approximate string queries using edit distance for illustration purposes, but our techniques also work for many types of set-similarity queries.

### A. Pruning with Filters

Pruning candidate answers based on the $T$ lower bound as discussed previously is referred to as the "count filter" [10]. We briefly review a few of the many other important filters.

**Length Filter:** An answer to a query with string $s$ and edit distance $k$ must have a length in $[|s| - k, |s| + k]$ [10]. Analogous findings exist for other similarity functions [15].

**Prefix Filter:** Suppose we impose a global ordering on the universe of grams. Consider a query with an ordered gram set $G(r)$ and lower bound $T$. An answer must share at least one gram with the query in its "prefix". That is, the first $|G(r)| - T + 1$ grams of an answer $r$ must share at least one gram with the first $|G(r)| - T + 1$ grams of the query [9].

**Other Filters:** Some filters exploit the the positions of $g$-grams, e.g., the position filter [10] and content-based filter [20], or mismatching $q$-grams [20] for pruning.

### B. Filter Tree

We can use filters to partition the data strings into groups. We use a structure called FilterTree [15] to facilitate such a partitioning. Originally, the FilterTree was designed for in-memory inverted lists. In Section III-B we discuss how to place the inverted lists onto disk.

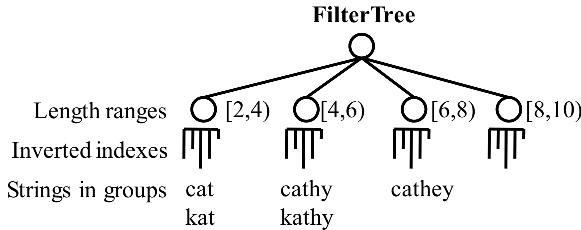



Fig. 2. A FilterTree partitioning data strings by length.

Figure 2 shows a FilterTree that partitions the data strings by their length. A leaf node corresponds to a range of lengths (called group), and each string belongs to exactly one group. For each group we build an inverted index on the strings in that group. To answer a query we then only need to process some of the groups. For example, the answers to a query with string "`cathey`" and edit distance 1 must be within the length range [5, 7]. To answer the query, we only access the inverted indexes of the *relevant* groups [4,6) and [6,8).

## III. Disk-Based Index

In this section we introduce a disk-based index to answer approximate string queries. We detail its individual components, and discuss how to construct and store them on disk.

### A. System Context

Suppose we have a table about persons with attributes such as address, name, phone number, as shown in Figure 3. We want to create an index on the "Name" attribute to support similarity selections. All components except the source table belong to this index and are intended to be part of the database engine. The components below the dotted line are stored on disk, and only the FilterTree is assumed to fit in memory.

**Dense Index**: The lowest level of our index is a dense index that maps from names to record ids in the source table. Each entry in the dense index is identified by a unique id, called string id (SID). Our decision to project the indexed column into a dense index is motivated by the following observations: (1) The number of candidates answers could be much higher than the number of true answers. Since the tuples in the source table could have many attributes, it would be inefficient to retrieve them for removing false positives. Instead, we use the dense index for post-processing and only access the source table for true answers. (2) The additional level of indirection allows us to choose the physical organization of the dense index independently of the source table, which can improve query performance (see Section III-D).

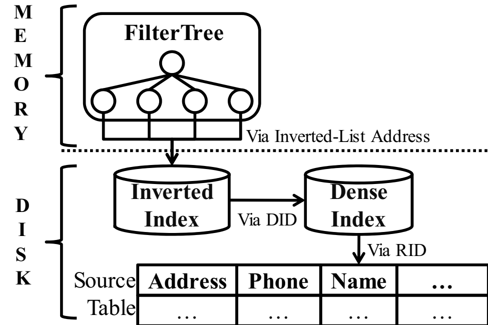



Fig. 3. Components of an index on the "Name" field of a "Person" table.

**Inverted Index and FilterTree**: The final level of our index consists of an in-memory FilterTree (Section II-B) and a corresponding disk-resident inverted index. Each leaf in the FilterTree has a map from each of its grams to an inverted list address $(l, o)$, indicating the offset $o$ and length $l$ of a gram's inverted list in a particular group. The inverted index stores inverted lists of string ids (SIDs) in a file on disk. Typically, the FilterTree is very small (a few megabytes) compared to the inverted index (hundreds of megabytes or gigabytes).

**Answering Queries:** To answer a query, we first identify all relevant groups with the FilterTree. Then, for each relevant group we retrieve the inverted lists corresponding to the query tokens, and solve the $T$-occurrence problem (Section II) to identify candidate answers. To remove false positives, we retrieve the candidates from the dense index and compute their real similarity to the query. In further discussions we ignore

the final step of retrieving the records from the source table since this step cannot be avoided, and we want to keep our solutions independent of the organization of the source table.

### *B. Partitioning Disk-Based Inverted Lists*

Inverted lists are often stored contiguously on disk [22] so each list can be read sequentially. This layout maximizes selection-query performance at the expense of more costly updates. As mentioned in Section II-A, we use a filter to partition the data strings into different groups, e.g., based on their length. For each group we build a $q$-gram inverted index. To answer a query, we process all *relevant* groups that could contain answers according to the partitioning filter used, e.g., groups that are within a certain length range.

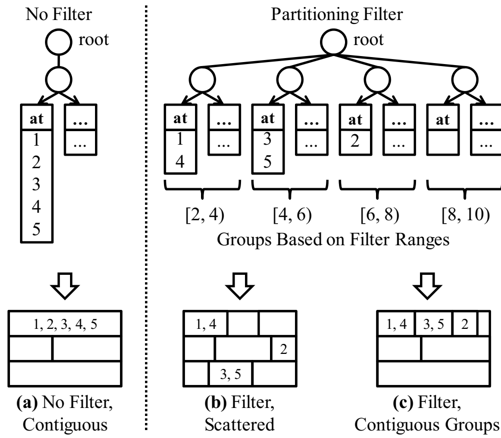



Fig. 4. The inverted list of gram "at" with and without partitioning and possible mappings to a file on disk. Organization (c) is the best.

Let us examine Figure 4 to discuss possible layouts of inverted lists partitioned in this fashion. In the left upper portion we see a gram at's inverted list without partitioning. To the right, the original inverted list is split into different groups according to a filter. At the bottom we show possible storage layouts. Without partitioning we can simply store the list contiguously as shown in organization (a). In organization (b), each inverted list is stored contiguously, but the lists belonging to the same gram from different groups are scattered across the file. Finally, organization (c) places the inverted lists belonging to the same gram contiguously in the file.

**Example:** Suppose we partitioned the strings by length, as shown in Figure 4. Consider a query with the string "cathey" and an edit-distance threshold $k = 1$. Let us examine the cost for retrieving the inverted list of gram at. In organization (a), we perform one disk seek and transfer the five elements $\{1, 2, 3, 4, 5\}$ for at's inverted list. With filtering, answers to the query must be within the length range $[5, 7]$, since the length of the query string "cathey" is six. Therefore, only the groups $[4, 6)$ and $[6, 8)$ are relevant. Using organization (b) we transfer three list elements $\{2, 3, 5\}$ with two disk seeks. Note that the elements $\{3, 5\}$ and $\{2\}$ could be arbitrarily far apart in the file. Using organization (c) we transfer the same three elements $\{2, 3, 5\}$ but with only one disk seek.

### *C. Constructing Inverted Index on Disk*

We now present how to efficiently build an inverted index in the physical layout shown in Figure 4(c). Since we want to index large string collections, we cannot assume that the final index fits into main memory. Hence, we desire a technique that (1) can cope with limited buffer space, and (2) scales nicely with increasing buffer space. Our main idea is to first construct an unpartitioned inverted index with a standard merge-based method [22] (Phase I), and then reorganize the inverted lists to reflect partitioning (Phase II). This two-phase approach has the advantage that any existing construction procedure for an inverted index can be directly used in Phase I, without modification. Later in this subsection, we discuss an alternative construction procedure that directly modifies a standard merge-based technique. Figure 5 shows both phases of index construction, described as follows.

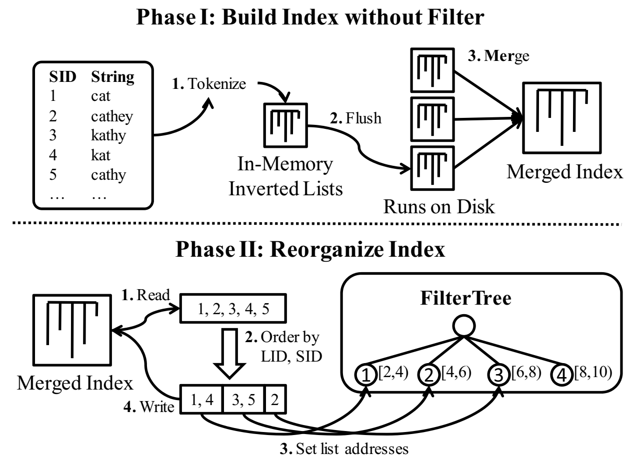



Fig. 5. Two phases of index construction. Phase I builds the index without partitioning. Phase II re-organizes the index to reflect partitioning.

**Phase I:** This is a standard merge-based construction procedure [22]. We read the strings in the collection one-by-one, decompose them into grams, and add the corresponding string ids into an in-memory inverted index. Whenever the in-memory index exceeds the buffer limitation, we flush all its inverted lists with their grams to disk, creating a run. Once all strings in the collection have been processed, we combine all runs into the final index in a merge-sort-like fashion.

**Phase II:** This phase reorganizes the inverted index constructed in Phase I into a partitioned inverted index better suited for approximate string queries. We start with an "empty" FilterTree, whose leaves contain empty mappings from grams to list addresses. We assign a unique id to each leaf (LID). Then, we sequentially read lists from the merged index until the buffer is full. For each of those lists, we re-order its elements based on their FilterTree leaf ids (LID) and string ids (SID). The re-ordering can be supported by a map from $SID$ to $LID$ created in Phase I, or we can compute the corresponding $LIDs$ on-the-fly to save memory. In the process, we set the lengths and offsets (list addresses) for the lists of different groups in the FilterTree. After the elements of each list in the buffer have been re-ordered, we write these lists in the buffer back to disk, in-place, in one sequential write. We

repeat these steps until all the lists of the merged index have been processed. Notice that the cost of Phase II is independent of the number of groups used in partitioning. If the merged index has $m$ bytes and our buffer holds $b$ bytes, then Phase II requires $2 * \frac{m}{b}$ sequential I/Os plus the CPU cost for sorting. The total cost is governed only by the index size and the buffer size, and thus Phase II meets our performance requirements.

**Integrating Partitioning in Phase I:** Instead of reorganizing an unpartitioned index in a separate phase (Phase II), we could directly modify the merge-based construction procedure to build a partitioned index as follows. When merging the runs on disk we immediately reorder the elements of a completely assembled inverted list before writing it to the final index, similar to the reordering done in Phase II. We also need to update the mappings in the FilterTree accordingly. The advantage of this construction procedure is that it eliminates the reorganization phase (Phase II). However, the procedure will need to be either written from scratch, or integrated into an existing merge-based construction algorithm.

### D. Placing Dense Index on Disk

Next we study different ways to organize the entries in the dense index. Recall that to answer a query we use the inverted lists to get candidate answers. Then, for each candidate id, we access the dense index to retrieve its corresponding <String, RID> pair and compute its similarity to the query (Section III-A). Intuitively, we want those entries that are likely to be accessed by the same query to be physically close on disk. That is, we want to minimize the number of blocks that need to be accessed to retrieve a given set of candidates. At the same time we want to avoid seeking through the entire dense-index file to find the candidates. Note that these desiderata are different, e.g., even if we only needed to retrieve a few blocks, those requested blocks could be far from each other in the file. The following are two natural ways to organize the entries:

**Filter Ordering:** Partitioning filters offer a simple but effective solution. If we sort the entries of the dense index by a filter, then we limit the range of blocks in which candidates of a query could appear. Also, it is more likely for blocks to contain entries that are similar to each other, and hence, similar to a query. For example, we can limit the blocks that could contain answers to a query "`cathey`" with edit distance 1 to those that contain strings of length [5,7]. Within that range we may perform random accesses to get the needed blocks, but the seek distances between these blocks are relatively small.

**Clustering:** Another solution is to group similar entries together in the file by running a clustering algorithm on them (e.g., k-Medoids). This global solution has drawbacks: (1) Clusters accessed by the same query could be arbitrarily far apart in the file. (2) The clustering process could be very expensive. (3) Updates to the index would be complicated. It is also possible to combine a filter ordering with clustering (hybrid), e.g., we could sort the strings by their length, and then we run a clustering algorithm within each length group.

We implemented all the above organizations, i.e., filter orderings, clustering using k-Medoids, and a hybrid approach. We experimentally found that all of the approaches perform much better than an arbitrary ordering of the strings. The more sophisticated methods, clustering and hybrid clustering, did not offer better query performance than a simple filter ordering. Therefore, we prefer the filter ordering solution.

### E. Index Maintenance

**Inverted Index:** Standard techniques [14] for inverted-index maintenance can be used to update our partitioned inverted index. They mostly involve buffering insertions into an in-memory index, and deletions into an in-memory list. Periodically the in-memory structures are integrated into the index on disk, e.g., using one of the following strategies [14]:

*In-Place*: For each list in the in-memory index, read the corresponding list from disk, combine the two lists, and write the new list back to disk (possibly to a different location).

*Re-Merge*: Read the lists in the file one-by-one. Whenever a list has pending changes in the in-memory index, update the list. Finally, write all (possibly updated) lists into a new file.

We can apply such maintenance strategies to our new physical layout from Section III-B. We add a FilterTree structure to the in-memory index that buffers updates. Alternatively, we could omit the FilterTree, and deal with reorganizing the inverted lists to reflect partitioning when we combine the in-memory index with the one on disk.

**Dense Index:** The dense index is a sorted sequential file which allows various implementations and associated maintenance strategies. For example, we could simply use a B+-tree. Since updates to the inverted index are done in a "bulk" fashion, it would be prudent to update the dense index (B+-tree) in a similar way to maintain the sequentiality of (leaf-) entries. For example, bulk-loading a new B+-tree or using an extend-based B+-tree would achieve this goal.

## IV. Cost-Based, Adaptive Algorithm

In this section we present a cost-based, adaptive algorithm for answering similarity queries using the disk-based indexes. For simplicity, we first describe the algorithm assuming the inverted-index organization in Figure 4(a), and later we discuss how to modify it to work with organizations (b) and (c).

### A. Intuition

Recall that to answer a query we tokenize its string into grams, retrieve their inverted lists, solve the $T$-occurrence problem to get candidate answers, and finally retrieve those candidates from the dense index to remove false positives. We develop the algorithm based on the following observations. (1) Candidate answers must occur on at least a certain number of the query grams' inverted lists, but not necessarily on all those inverted lists. (2) The pruning power of inverted lists comes from the *absence* of string ids on them. For example, an inverted list that contains all string ids cannot help us remove candidates. Intuitively, long lists are expensive to retrieve from disk and may not offer much pruning power. An approximate query with a string $r$, $q$-grams $G(r)$, and an edit distance threshold $k$ must share at least $T = |G(r)| - k * q$

grams with an answer, with $|G(r)| = |s| - q + 1$. According to the prefix filter (Section II-A), the minimum number of lists we must read to ensure the completeness of answers is $minLists = |G(r)| - (T - 1)$. To understand this equation, consider a string id that occurs on $T - 1$ lists. To become a candidate, it must additionally occur on at least one of the other $|G(r)| - (T - 1)$ lists. For example, if $q = 2$ and $k = 1$, a query with string "cathey" has $|G(r)| = 6 - 2 + 1 = 5$ grams, $T = 5 - 1 * 2 = 3$ and $minLists = 5 - (3 - 1) = 3$.

At one extreme, we could read just the $minLists$ shortest lists. As a consequence, the number of candidates for post-processing could be high, because every string id on those lists needs to be considered. Recall that for post-processing we retrieve strings from the dense index (Figure 3), hence every candidate could require a random disk I/O. At the other extreme, we could read all the inverted lists for a query, including those long and expensive ones with little pruning power. It is natural to strive for a solution between the two extremes, balancing the I/O costs for retrieving inverted lists and for probing the dense index.

### *B. Algorithm Details*

To answer a query we begin by reading the $minList$ shortest inverted lists (corresponding to grams in the query) from disk into memory. Then we traverse these lists to obtain a set of initial candidates containing all the lists' string ids (using an algorithm such as "HeapMerge" [18]). The set of initial candidates, denoted by $C = \{c_1, c_2, \ldots, c_{|C|}\}$, is a set of triples $c = (sid, cnt, cnt_a)$. In each triple, $sid$ denotes the string id of the candidate, $cnt$ denotes the current "count" or the total number of occurrences of $sid$ on the lists we have read so far, and $cnt_a$ denotes the "count absent" value (its meaning will become clear shortly). Next, we decide whether we can reduce the overall cost of the query by reading additional lists to prune candidates. If so, then we read the next list and prune candidates with it. If not, then we post-process the candidates in $C$. The decision whether or not to read the next list depends on the current number of candidates still in $C$, the cost for post-processing them by accessing the dense index, the cost for reading more lists, and the likelihood of pruning candidates by reading more lists. We repeat this process until we have read all the query grams' inverted lists or there is no cost reduction from reading additional inverted lists.

Note that the benefits of reading additional inverted lists might manifest themselves after reading a few more lists, and not necessarily after reading only one more list. Hence, we must estimate the effects of reading the next $\lambda$ lists. Take our running example, with $r$ = cathey, $k = 1$, and lower bound $T = 5$ as shown in Figure 6. Let us examine candidate $c_1 = (2, 2, 2)$, which is part of the initial candidates produced after processing the $minLists = 3$ shortest lists. We have read 3 of the 5 lists, so there are 2 lists left. In order to prune candidate $c_1$, its string id must be absent on at least $cnt_a = 2$ additional lists.

In general, the "count absent" value is the minimum number of additional lists a candidate string id must be *absent* from,

**Inverted Lists for Query s=cathey, k=1, q=2, T=3**

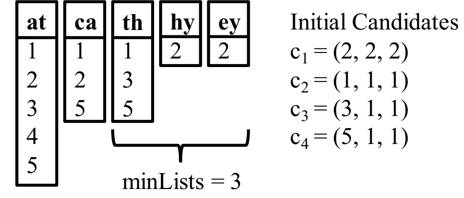



Fig. 6. Example to illustrate meaning and effect of the "count absent" value.

in order to be pruned. At each stage in the adaptive algorithm for answering a query with string $r$ and $q$-grams $G(r)$, we can compute the count-absent value of a triple $(sid, cnt, cnt_a)$ corresponding to a candidate answer by:

$$cnt_a = (|G(r)| - minLists) - (T - cnt) + 1. \quad (1)$$

Candidates could have various $cnt_a$ values, and the cost and benefit of reading more lists depend on $\lambda$, i.e., the number of additional lists we consider reading. Since the benefits of reading more lists can become apparent only after reading several next lists, a cost model that considers only reading the next list could get stuck in a local minimum, and is therefore insufficient to find a globally minimal cost. This is the reason why we need to consider reading all the possible numbers of remaining lists $\lambda$. For example, in the first iteration we would consider $1 \leq \lambda \leq |G(r)| - minLists$.

**Pseudo-Code:** Figure 7 shows pseudo-code for the adaptive algorithm. We use $\Omega(r)$ to denote the list addresses sorted by length in an increasing order for a query string $r$. For convenience we use $\Omega(r)[j]$ to mean the $j$-th list in $\Omega(r)$. $C_\lambda$ refers to the subset of candidates with $cnt_a \leq \lambda$. We assume the following parameters of a cost-model: (1) $\Theta$ is the average cost of post-processing a candidate, (2) $\Pi(l)$ is the cost of reading list $l$, and (3) $ben(\lambda)$ is the benefit or reading $\lambda$ additional lists.

```
Input: Inverted list addresses Ω(r) for a query string r
       Minimum number of lists to read minLists
Output: A set C of candidate answers to be post-processed
Method:
1.  C = HeapMerge(Ω(r), minLists);
2.  nextList = minLists + 1;
3.  listsLeft = |Ω(r)| − minLists;
4.  WHILE listsLeft > 0
5.      FOR λ = 1 TO listsLeft
6.          Λ = next λ lists starting from Ω(r)[nextList];
7.          costPP = |C_λ| ∗ Θ; // cost of post-processing
8.          costRL = Σ_{l∈Λ} Π(l); // cost of reading lists
9.          benfitRL = ben(λ) for lists in Λ;
10.         IF (costRL - benefitRL < costPP) THEN
11.             invList = readList(Ω(r)[nextList]);
12.             C = pruneCandidates(C, invList);
13.             nextList+=1; listsLeft−=1;
14.             BREAK FOR;
15.         END IF
16.     END FOR
17.     IF λ == listsLeft THEN BREAK WHILE; // no benefit
18. END WHILE
19. RETURN C;
```

Fig. 7. Cost-based, adaptive algorithm for answering queries.

We begin by reading $minLists$ inverted lists and process them to obtain a set of initial candidates (line 1). Next, we consider reading all numbers of remaining lists, setting $\lambda$ accordingly. We compute the cost of post-processing the candidates that could potentially be pruned with $\lambda$ lists (line 7). Correspondingly, we compute the cost (line 8) and benefit (line 9) of reading the next $\lambda$ lists. We decide whether we can reduce the overall cost by reading $\lambda$ lists, as compared to not reading them (line 10). If we can reduce the cost, we read the next list (line 11) and prune candidates (line 12). Otherwise, we proceed with the next $\lambda$. We repeat this process until (a) we have read all lists or (b) we cannot benefit from reading any more lists. Notice that whenever we detect a benefit we read just *one* more list at a time, independent of the value of $\lambda$. This approach mitigates possible inaccuracies of a real cost-model.

### C. Cost Model

In this section we develop a model to estimate the cost and benefit of reading additional inverted list.

**Cost of post-processing candidates**: We focus on the I/O cost because the CPU time for computing similarities is often negligible in comparison. The maximum cost for retrieving a candidate string is the time for one disk seek plus the time to transfer one disk block. This cost can be determined directly from the hardware configuration or estimated offline. We can improve this model by taking into account the chance that a dense index block is already cached in memory. Using a conservative but simple approach, we read a few blocks offline and compute the average time to retrieve a block, denoted by $\Theta$. We then use $|C| * \Theta$ to estimate the cost of post-processing a candidate set $C$. For example, the cost of post-processing the candidates in Figure 6 would be 4*$\Theta$ since $|C| = 4$.

**Cost of retrieving inverted lists**: Since the lengths of inverted lists usually follow a Zipf distribution, the average retrieval time is not an accurate estimator of the true cost. Again, the cost is determined by the seek time and the transfer rate of the disk. Since our solution is on top of a file system, the raw disk parameters are not very accurate performance indicators either, due to the intermediate layer. To overcome this issue, we read a few lists offline and do a linear regression to obtain the cost function $\Pi(\omega) = m * \omega.l + b$, where $\omega.l$ denotes the length of the inverted list $\omega$, $b$ is an estimate for the seek time, and $m$ is an estimate for the inverse transfer rate. So, given a set $\Omega$ of $\omega$, we can estimate the total cost of reading all those inverted lists by $\sum_{\omega \in \Omega} \Pi(\omega)$. For example, the cost of reading the lists of grams `ca` and `at` from Figure 6 would be $(m * 3 + b) + (m * 5 + b)$, since their lengths are 3 and 5, respectively.

**Benefit of reading additional inverted lists**: At a high level, we quantify the benefit in terms of the cost we can save by pruning candidates, considering the likelihood of that happening by reading $\lambda$ lists. The likelihood of pruning a candidate $c$ with $\lambda$ lists depends on its $cnt_a$. Intuitively, we would like to know the probability that $c$'s string id is absent from at least $cnt_a$ of the $\lambda$ lists. Computing this probability for each candidate individually would be computationally expensive. To avoid repeated computations we group the candidates by $cnt_a$ as follows. We define a subset of $C$ as $C(i) = \{c | c \in C \wedge c.cnt_a = i\}$, containing all candidates in $C$ that have a certain $cnt_a = i$. We also make the following simplifying assumptions:

- The probability of one string id being present or absent from an inverted list is independent of the presence or absence of another string id on the same list.
- The probability of a string id being present or absent from one inverted list is independent of that same string id being absent or present from another list.

Following these assumptions, we estimate the benefit as:

$$ben(\lambda) = \Theta * \sum_{i=1}^{\lambda} |C(i)| * p(i, \lambda), \quad (2)$$

where $p(i, \lambda)$ denotes the probability of a string id being absent from $i$ of the $\lambda$ lists. In a sense, we are being optimistic, since Equation 2 expresses that we could prune *all* candidates in $C(i)$ with probability $p(i, \lambda)$. The key challenge is to obtain a reasonably accurate $p(i, \lambda)$ efficiently. Following our assumptions we model $p(i, \lambda)$ as a sequence of $\lambda$ independent Bernoulli trials, i.e., a Bernoulli process. We define "success" as the absence of a string id on a list, and "failure" as the presence of a string id on a list. Since all we know of the lists are their lengths, we estimate the success probability with a set $\Omega$ of list addresses $\omega$ as:

$$p_s = 1 - \frac{\sum_{\omega \in \Omega} \omega.l}{|\Omega|} * \frac{1}{N}. \quad (3)$$

The term $\frac{\sum_{\omega \in \Omega} \omega.l}{|\Omega|}$ denotes the average list length of the $\lambda$ lists in $\Omega$ we are considering to read, and $N$ is the total number of strings in the dense index. The probability of at least $i$ successes in $\lambda$ trials having a success probability $p_s$ can be computed with the cumulative binomial distribution function:

$$p(i, \lambda) = binom(i, \lambda, p_s) = \sum_{j=i}^{\lambda} \binom{\lambda}{j} p_s^j (1 - p_s)^{\lambda - j}. \quad (4)$$

Combining Equations 2, 3 and 4 yields a complete model for estimating the benefit of reading $\lambda$ additional lists:

$$ben(\lambda) = \Theta * \sum_{i=1}^{\lambda} |C(i)| * binom(i, \lambda, p_s). \quad (5)$$

### D. Combining with Partitioned Inverted-Index Layout

So far, we have presented the adaptive algorithm using the layout in Figure 4(a) from Section III for simplicity. Next, we discuss how to combine the adaptive algorithm with the other two inverted-index layouts (b) and (c). Recall that when answering a query we first traverse the FilterTree to identify the relevant groups (e.g., corresponding to length ranges) which could contain answers.

**Modified Success Probability:** In Equation 3 the probability (called success probability) of a string id being absent from a set of lists depends on their average list length, and

the total number of strings $N$ in the dataset. Since each group generated by a partitioning filter only contains a subset of the $N$ total strings, simply applying Equation 3 would lead to overestimation of the success probability, and consequently overestimation of the benefit (Equation 5). We modify the success probability by replacing $N$ in Equation 3 with the number of strings in the particular group we are considering. This change applies to both inverted-index layouts (b) and (c). Next, we detail how to answer queries with those two layouts.

**Layout (b):** To answer a query using the adaptive algorithm, we first identify all relevant groups, and then process each group individually as if there was no partitioning (but with the new success probability). Each instance of the algorithm tries to minimize the cost for processing its corresponding group.

**Layout (c):** Intuitively, we just process all relevant groups together, as opposed to processing them one-by-one. We first read the $minList$ inverted lists of all relevant groups to create the sets of initial candidates for *all* groups at the same time. To exploit the contiguous layout, we retrieve the lists gram by gram. That is, we read the lists of all relevant groups of the first gram, then we read the lists of all relevant groups of the second gram, etc., until we have retrieved all the $minLists$ lists. We perform HeapMerge on the groups separately to get the initial sets of candidates.

Next, we proceed with the iterative phase of the algorithm, estimating the cost and benefit of reading the next grams' lists of all relevant groups. We either decide to read the next lists together or commence post-processing.

**Layout (c) Examples:** In the following, we describe how to handle a few interesting scenarios when combining the adaptive algorithm with organization (c). For example, Figure 8 shows the inverted lists of the relevant groups for our running example `cathey` (the string ids differ from earlier examples). To minimize the global cost we should read the lists for gram `hy` first because there are a total of 3 relevant elements $\{2, 6, 8\}$. Next, we should read the lists of gram `ey` because they contain a total of 4 elements $\{2, 6, 7, 8\}$, and so on.

*Local vs. Global Ordering:* We want to read the inverted lists from shortest to longest. However, the local order of each group may not correspond to the best global order. We must read the relevant lists of grams in order of their sum of lengths.

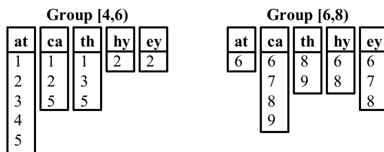



Fig. 8. Inverted lists for query `cathey`, $k = 1$, $q = 2$, and $T = 3$.

*Pruning Entire Groups:* Groups that do not contain at least $T$ grams of the query can be pruned entirely. We should not read the lists that do match a query's grams of a pruned group.

Figure 9 shows a group [6,8) that has less than $T$ of the query's grams. Since this group cannot contain answers to the query, we do not need to read lists from it. For example, for gram `ca` that exists in group [6,8), we should only read the elements $\{$1,2,5$\}$ and *not* the elements $\{$6,7,8,9$\}$.

A similar situation can arise during the iterative phase of the adaptive algorithm. When a particular group does not contribute candidates anymore (because they have all been pruned), then we do not need to read inverted lists from that group anymore (but possibly still from the other ones). We must handle both scenarios above specially during cost estimation and during the retrieval of lists.

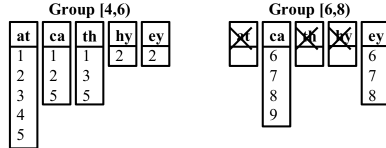



Fig. 9. Inverted lists for query `cathey`, $k = 1$, $q = 2$, and $T = 3$. None of the strings in group [6,8) contain the grams `at`, `th`, or `ty`.

## V. Experiments

In this section, we evaluate the performance of queries and index construction of our techniques. We show experiments on range and top-k queries using edit distance, and range queries using normalized edit distance, Jaccard based on $q$-grams, and Jaccard based on word grams. We compare our techniques to a recent tree-based index BED-tree [21], whenever possible.

**Datasets:** We used six real datasets, summarized in Table I. The first four datasets are taken from [21] (BED-tree) to establish a fair comparison. In addition, we used the last two datasets for experiments on scalability and index-construction since they are larger than the first four.

TABLE I
Datasets and their statistics.

| Dataset | # Strings | Max Len | Avg Len |
|---|---|---|---|
| DBLP Author | 2,948,929 | 48 | 15 |
| DBLP Title | 1,158,648 | 667 | 68 |
| IMDB Actor | 1,213,391 | 73 | 16 |
| Uniprot | 508,038 | 1992 | 341 |
| Medline Titles | 10,000,000 | 252 | 84 |
| Web Word Grams | 20,000,000 | 163 | 23 |

The DBLP Title and Author datasets[1] were taken from DBLP, and contained authors and titles of publications. The IMDB Actor[2] dataset consists of actor names from movies. The Uniprot[3] dataset contains protein sequences in text format. The Medline Titles[4] dataset consists of publication titles from the medical domain. Finally, the Web Word Grams[5] dataset contains popular word-grams from web documents. We randomly picked 10 million titles of the Medline Dataset, and 20 million 4-word-grams for the Web Word Grams dataset.

**Hardware and Compiler:** We conducted all experiments on a machine with a four-core Intel Xeon E5520 2.26Ghz processor, 12GB of RAM, and a 10,000 RPM disk drive, running a Ubuntu operating system. We used the original code for BED-tree written in C++ which the authors generously

[1] www.informatik.uni-trier.de/~ley/db
[2] www.imdb.com
[3] www.uniprot.org
[4] www.ncbi.nlm.nih.gov/pubmed
[5] www.ldc.upenn.edu/Catalog, number LDC2006T13

provided to us. We implemented all our algorithms in C++ as well. We compiled all code with GCC using the "-O3" flag.

**Parameters:** We experimented with different $q$ for tokenizing strings into $q$-grams and found $q = 3$ to be best for most cases. Therefore, we used $q = 3$ for all experiments, for both BED-tree (where applicable) and for out techniques. For our techniques, we used the length filter for partitioning. We used a disk-block size of 8KB for both BED-tree and our methods.

**Clearing Filesystem Cache:** In our experiments we considered both the raw disk performance of queries, and their performance with caching. To simplify the implementations, both BED-tree and our techniques were built on top of a filesystem (as opposed to using the raw disk device). Using a filesystem, however, complicates accurate measurements of disk performance due to filesystem caching (it will aggressively use all available memory to cache disk pages). To overcome this issue, we cleared the filsystem cache at certain points (to be explained) with the following shell command:

```
echo 3 > /proc/sys/vm/drop_caches
```

**Query Workloads:** The authors of the BED-tree provided us with the data and workloads from their experiments in [21]. The workload for each dataset consisted of 100 randomly chosen strings. For the other datasets used only in this paper (Medline Titles, Web Word Grams), we also generated workloads by randomly choosing 100 strings for each dataset.

### A. Index Construction Performance

We built inverted indexes for the Web Word Grams datasets in organization (c) (Section III) using length filtering. We measured each step of the construction procedure (1) creating runs, (2) merging the runs, and (3) reorganizing the index, clearing the filesystem cache before each step. We also ran this experiment on the Medline Titles dataset, but omit the results since they show a similar trend.

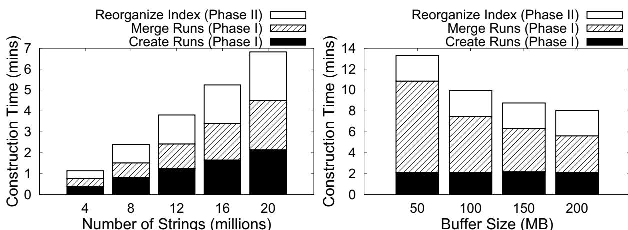



Fig. 10. Index construction performance on Web Word Grams.

In the left chart of Figure 10 we allocated a fixed buffer size of 400MB for index construction. It shows that the index-construction procedure scaled well (almost linearly) with the size of the dataset. The right chart shows the construction performance with varying buffer sizes, and we see that the merging of the runs took most of the time. By increasing the buffer size we improved the performance of merging the runs. The other two phases, creating the runs and reorganizing the index, did not benefit from a larger buffer size because they were CPU bound, explained as follows. Creating the runs consists of tokenizing the strings, and frequently reallocating in-memory inverted lists. Reorganizing the index consists of sorting inverted-list elements, while performing disk operations in buffer-sized chunks.

We do not report the construction times for BED-tree for the following reason. A comparison would be somewhat unfair since our technique builds its inverted index in "bulk", while BED-tree does not currently implement bulk-loading (it uses repeated insertions). For example, constructing a BED-tree on 20 million Web Word Grams with a 100MB buffer took around 9 hours, and still almost 4 hours with a 400MB buffer.

### B. Query Performance Naming Conventions and Methodology

In this subsection, we introduce the different flavors of BED-tree and the inverted-index approach used in our experiments on query performance. We also detail our procedures for obtaining the results of different types of experiments.

**BED-Tree Naming:** We follow the convention from [21]. BD, BGC, and BGL refer to a BED-tree using the dictionary order, gram count order, and gram location order, respectively.

**Naming of Our Approaches:** In our experiments we focused on the following two extreme approaches showing the best and worst inverted-index solutions for raw disk performance. "Simple" refers to a straightforward adoption of existing algorithms. It uses an unpartitioned inverted index and a dense index whose entries are in an arbitrary order. "Simple" retrieves all the inverted lists of a query string's grams and then solves the $T$-occurrence problem with an efficient in-memory algorithm (we used DivideSkip [15]). We use "AdaptPtOrd" to refer to our most advanced method using the adaptive algorithm ("Adapt"), a partitioned inverted index ("Pt"), and a dense index with entries ordered by their length ("Ord"). More results exploring the various dimensions of our solutions can be found in the full paper [6].

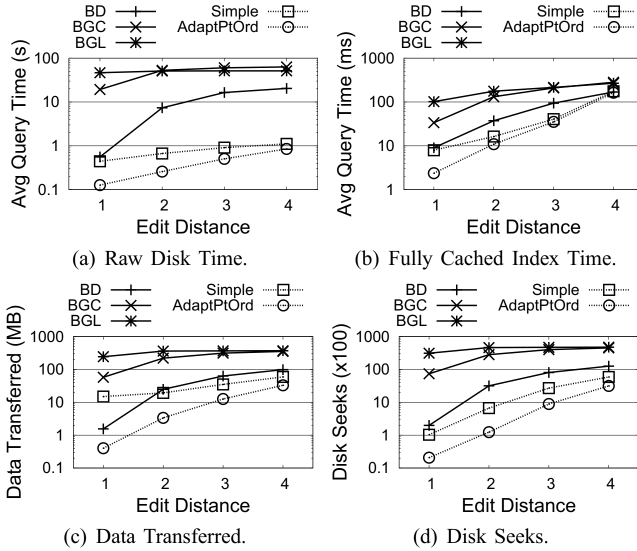
**Raw Disk:** In this type of experiments, we measured the performance of queries when all data required for answering a query (inverted lists, dense index blocks, BED-tree blocks) needed to be retrieved from disk. To do so, we cleared the filesystem cache before each query. Recall that our inverted-index assumes the FilterTree is in memory (Section III-A). For a fair comparison, we allocated the same amount of memory needed for the FilterTree to BED-tree's buffer manager. Note that BED-tree implements its own buffer manager, and therefore, those blocks cached in its buffer manager were unaffected by clearing the filesystem cache.

We also gathered the number of disk seeks and the amount of data transferred from disk per query captioned as "Data Transferred" and "Disk Seeks". For BED-tree the disk seeks are the number of nodes retrieved from disk (not already in the buffer manager), and the data transferred is that number multiplied by the block size. For our inverted-index solution, the number of disk seeks is the number of inverted lists and dense-index blocks accessed. We computed the data transferred using the sizes of the inverted lists and dense-index blocks.

**Fully Cached Index:** This experiment represents the other extreme in which all data required to answer a query is already in memory. For BED-tree we achieved this behavior

by allocating a large amount of memory in its buffer manager. We ran our workloads immediately after building the BED-tree, and therefore, the entire BED-tree was in memory when running queries. For our inverted-index approach we relied on the filesystem for caching. We first built the inverted index and dense index without clearing the filesystem cache, and then immediately ran our workloads, assuming that after construction all indexes are probably in the filesystem's cache.

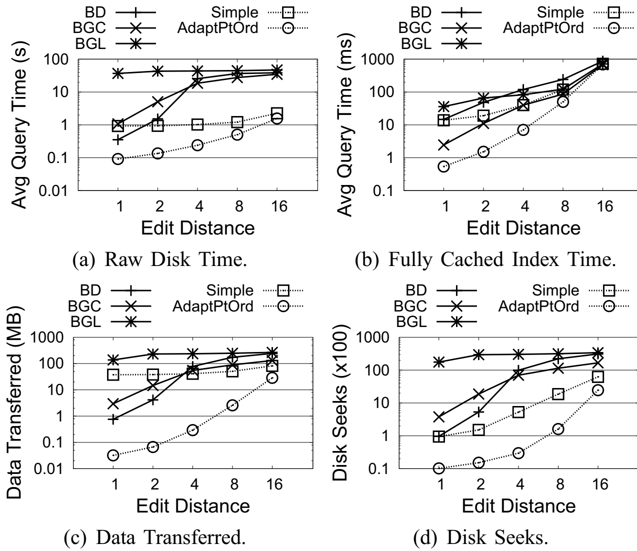
### *C. Range Queries Using Edit Distance*




(a) Raw Disk Time. (b) Fully Cached Index Time.

(c) Data Transferred. (d) Disk Seeks.

Fig. 11. Range-query performance on DBLP Author.

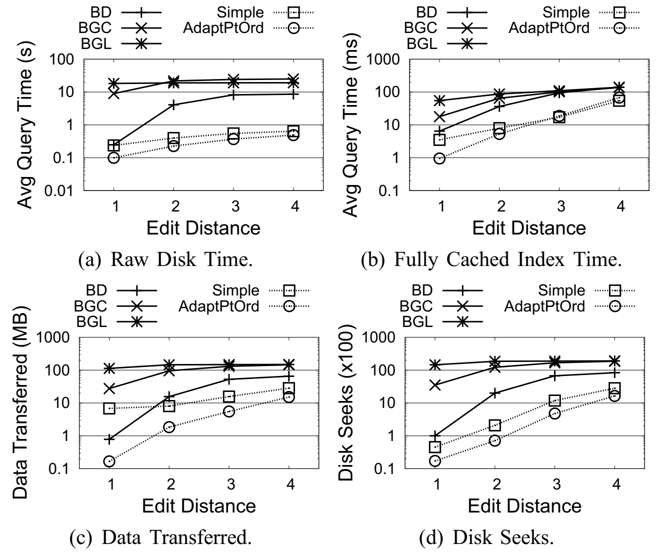



(a) Raw Disk Time. (b) Fully Cached Index Time.

(c) Data Transferred. (d) Disk Seeks.

Fig. 12. Range-query performance on DBLP Title.

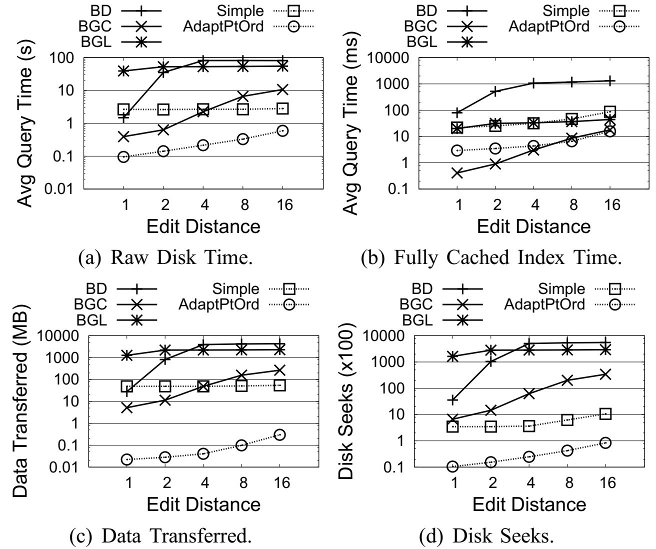
*1) Comparison with BED-Tree:* Next, we compare our approach with BED-tree on range-query performance using the datasets and workloads from the BED-tree paper [21]. The first two graphs, (a) and (b), of Figures 11-14 show the raw disk, and fully cached index times, respectively. The graphs (c) and (d) further detail the raw disk performance with the average number of disk seeks and data transferred per query.




(a) Raw Disk Time. (b) Fully Cached Index Time.

(c) Data Transferred. (d) Disk Seeks.

Fig. 13. Range-query performance on IMDB Author.




(a) Raw Disk Time. (b) Fully Cached Index Time.

(c) Data Transferred. (d) Disk Seeks.

Fig. 14. Range-query performance on Uniprot.

We see that "AdaptPtOrd" consistently outperformed BED-tree by orders of magnitude, both for raw disk peformance and for a fully cached index (notice we are using a log scale on the y-axes). Also, "AdaptPtOrd" was considerably faster than "Simple". The performance differences between "AdaptPtOrd" and the BED-tree variants on raw disk are explained by the graphs (c) and (d). "AdaptPtOrd" transferred significantly less data per query with fewer disk seeks than BED-tree. The two main reasons why BED-tree examined so many nodes are as follows. First, the pruning power at higher levels of the BED-tree is weak because a node entry refers to the enclosing interval of ranges in its subtree. Second, the BED-tree search procedure traverse multiple paths in the tree (more akin to an R-tree search), leading to additional node accesses as compared to a standard B-tree range search. Such a search procedure can also incur long disk-seek distances, because it is impossible to simultaneously store all tree-nodes close to each other (in a standard B-tree we only need to store the leaves close to their siblings). Our argument that BED-tree's pruning

power is not as strong as our approach is also supported by the results on fully cached indexes, where a significant cost is computing the real edit distances to candidate answers.

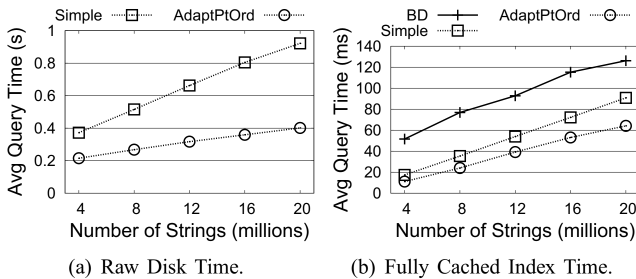
In Table II we summarized the index sizes for these sets of experiments. We observe that, in general, our inverted index approach requires more space than BED-tree. For example, on DBLP Author the BED-tree with dictionary ordering (BD) required 97MB of disk-space, and our indexes required 82 + 204 = 292MB on disk. However, our approaches transferred much less data from disk per query (see Figures 11-14). Also, recall that for the raw disk experiments, we give the BED-tree variants a buffer space equal to "FT", the size of our FilterTree.

TABLE II
INDEX SIZES IN MB OF BED-TREE VARIANTS AND OUR INVERTED-INDEX COMPONENTS. DENIX REFERS TO THE DENSE INDEX, INVIX TO THE INVERTED INDEX, AND FT TO THE IN-MEMORY FILTERTREE.

| Dataset | BD | BGC | BGL | DenIx | InvIx | FT |
|---|---|---|---|---|---|---|
| DBLP Author | 97 | 225 | 189 | 82 | 204 | 7 |
| DBLP Title | 123 | 156 | 157 | 100 | 297 | 21 |
| IMDB Actor | 38 | 88 | 75 | 32 | 88 | 11 |
| Uniprot | 283 | 302 | 305 | 222 | 617 | 49 |

*2) Scalability:* In Figures 15 and 16 we varied the number of indexed strings on our two large datasets, Web Word Grams and Medline Titles, to evaluate the scalability of our techniques. Due to its slow performance we omit BED-tree from the raw disk experiments. For example, on 12 million Web Word Grams its best version BD needed an average of 15 seconds per query, and on 6 million Medline Titles its best version BGC needed an average of 145 seconds. Similarly, we only plot the best version of BED-tree for the in-memory results since the other versions were significantly worse.

Our results show that "AdaptPtOrd" offers better scalability than "Simple", explained as follows. As we increased the size of the dataset, some inverted lists became longer. However, the number of results per query grew relatively slower than the total index size. Especially for highly selective queries, the adaptive algorithm avoided reading many unnecessary inverted lists. This effect explains the excellent performance on the highly selective Medline Titles. Similar arguments hold for the experiments with fully cached indexes.

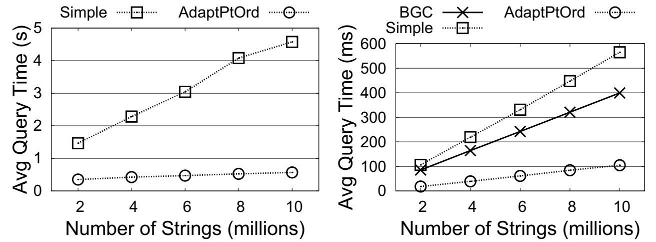



(a) Raw Disk Time. (b) Fully Cached Index Time.

Fig. 15. Range-query scalability with edit distance 2 on Web Word Grams.

## D. Top-K Queries Using Edit Distance

Next, we discuss our results on top-k queries shown in Figures 17 and 18. As before we used the datasets and

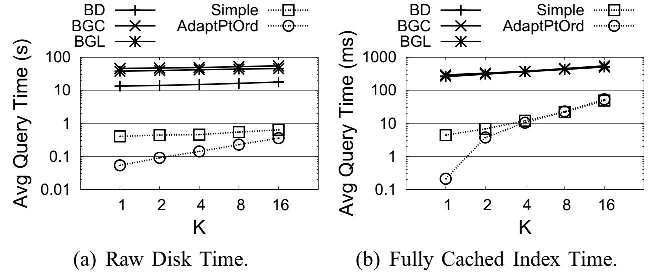



(a) Raw Disk Time. (b) Fully Cached Index Time.

Fig. 16. Range-query scalability with edit distance 6 on Medline Title.

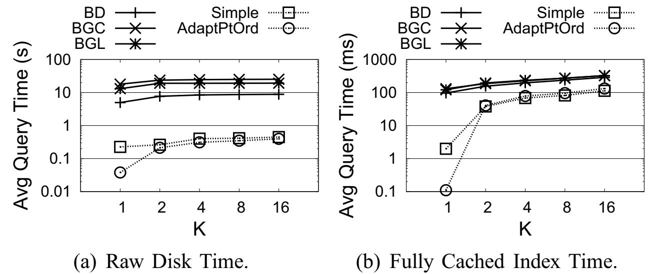
workloads of the BED-tree paper. We answer top-k queries on our inverted-index by a sequence of range queries with increasing ranges. We do not show the results on the DBLP Title and Uniprot datasets, because for some queries BED-tree did not find all correct answers. For example, on Uniprot with K=4, BED-tree only returned a total of 304 answers instead of the correct 400 answers for the 100 queries.

The results on top-k queries are consistent with those on range queries, and similarity show our techniques answered top-k queries efficiently, and outperformed BED-tree.




(a) Raw Disk Time. (b) Fully Cached Index Time.

Fig. 17. Top-K query performance on DBLP Author.



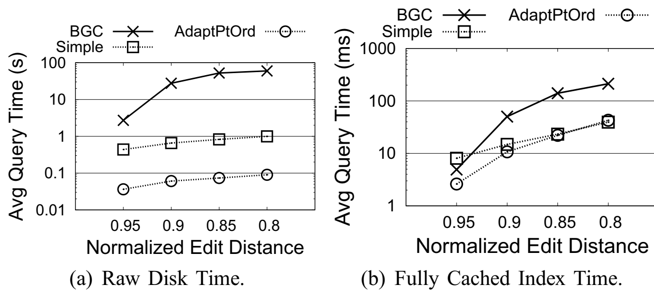

(a) Raw Disk Time. (b) Fully Cached Index Time.

Fig. 18. Top-K query performance on IMDB Actor.

## E. Query Performance Using Other Similarity Functions

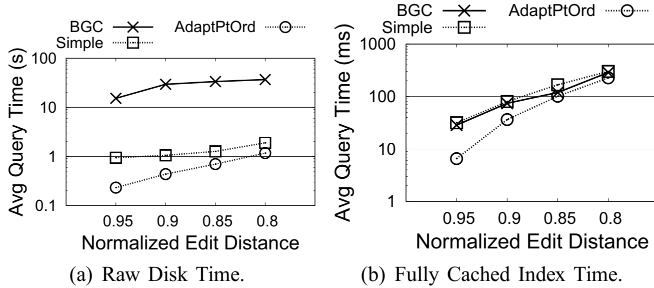
In this subsection, we present experiments on range queries using normalized edit distance and Jaccard. Its main purpose is to demonstrate that our solutions also support similarity functions other than edit distance. Due to space limitations, we omit some of the figures since they are consistent with the overall trends. The complete set of experimental results can be found in the full version of this paper [6].

*1) Normalized Edit Distance:* Again, we used the datasets and workloads from the BED-tree paper. Since BED-tree currently supports normalized edit distance only with the gram counting order, we have only BGC in Figures 19 and 20. As before, "AdaptPtOrd" outperforms its competitors.




(a) Raw Disk Time. (b) Fully Cached Index Time.

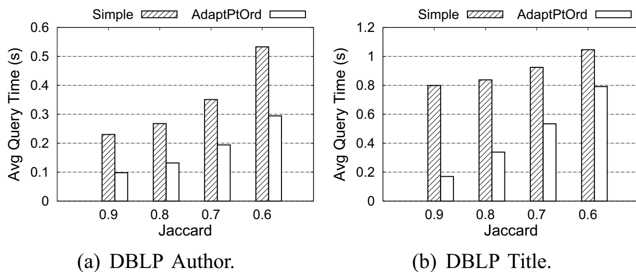
Fig. 19. Range-query performance on DBLP Author.




(a) Raw Disk Time. (b) Fully Cached Index Time.

Fig. 20. Range-query performance on DBLP Title.

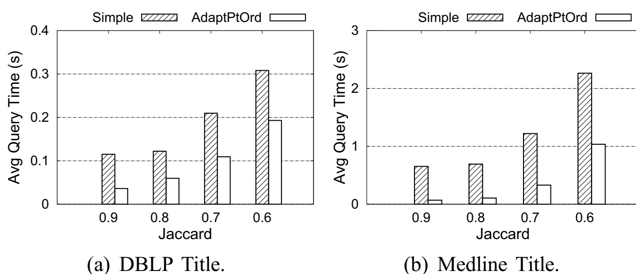
*2) Jaccard with Q-Gram Tokens:* In Figure 21 we used the Jaccard similarity of multisets of $q$-gram tokens to quantify the similarity between strings. Though BED-tree could possibly answer queries using Jaccard with the gram count ordering, its current implementation does not support it. Therefore, we only plot the results of our approaches. We observe that our new techniques also provide a disk-performance benefit to queries using Jaccard.




(a) DBLP Author. (b) DBLP Title.

Fig. 21. Raw disk range-query performance using Jaccard on $q$-gram tokens.

*3) Jaccard with Word Tokens:* For those datasets with very long strings (DBLP Title, Medline Title), it could be more meaningful to use Jaccard based on word tokens to quantify the similarity between strings. Figure 22 shows the raw disk performance of queries based on word tokens, and we see that our techniques also improved their performance in this setting.




(a) DBLP Title. (b) Medline Title.

Fig. 22. Raw disk range-query performance using Jaccard on word tokens.

## VI. Conclusion

We have studied approximate string selection queries when data and indexes reside on disk. We proposed a new physical layout for an inverted index, demonstrated how to efficiently construct it, and showed its benefits to query processing. We developed a cost-based adaptive algorithm to answer queries. We have shown that the adaptive algorithm and the new index layout complement each other and that their combination answers queries efficiently. Further, our techniques outperformed a recent tree-based index, BED-tree.

**Acknowledgements**: This work was supported by the CluE (IIS 0844574) and Asterix (IIS 0910989) NSF grants, and the National Nature Science of China grant number 60828004.

## References

[1] *Oracle Text, An Oracle Technical White Paper*, 2007. http://www.oracle.com/technology/products/text/pdf/11goracletexttwp.pdf.

[2] *Fuzzy Search in IBM DB2 9.5*, 2008. http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp?topic=/com.ibm.db2.luw.admin.nse.topics.doc/doc/t0052178.html.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, 2007.

[5] A. Behm, S. Ji, C. Li, and J. Lu. Space-constrained gram-based indexing for efficient approximate string search. In *ICDE*, 2009.

[6] A. Behm, C. Li, and M. J. Carey. Answering approximate string queries on large datasets using external memory (full version). Technical report, Department of Computer Science, UC Irvine, July 2010.

[7] S. Buettcher and C. L. A. Clarke. Index compression is good, especially for random access. In *CIKM*, 2007.

[8] S. Chaudhuri, K. Ganjam, V. Ganti, R. Kapoor, V. Narasayya, and T. Vassilakis. Data cleaning in Microsoft SQL Server 2005. In *SIGMOD*, 2005.

[9] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[10] 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.

[11] M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava. Fast indexes and algorithms for set similarity selection queries. In *ICDE*, 2008.

[12] E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In *In VLDB*, pages 139–148, 2001.

[13] Y. Kim, K.-G. Woo, H. Park, and K. Shim. Efficient processing of substring match queries with inverted q-gram indexes. In *ICDE*, 2010.

[14] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4), 2006.

[15] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[16] C. Meek, J. M. Patel, and S. Kasetty. Oasis: An online and accurate technique for local-alignment searches on biological sequences, 2003.

[17] G. Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[18] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, 2004.

[19] E. Ukkonen. Approximae string matching with q-grams and maximal matching. *Theor. Comut. Sci.*, 1:191–211, 1992.

[20] C. Xiao, W. Wang, and X. Lin. Ed-join: An efficient algorithm for similarity joins with edit distance constraints. In *VLDB*, 2008.

[21] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, 2010.

[22] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[23] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. 2006.