Keyword Query Cleaning

Ken Q. Pu
Faculty of Science
University of Ontario Inst. of Technology
ken.pu@uoit.ca

Xiaohui Yu
School of Information Technology
York University
xhyu@yorku.ca

ABSTRACT

Unlike traditional database queries, keyword queries do not adhere to predefined syntax and are often dirty with irrelevant words from natural languages. This makes accurate and efficient keyword query processing over databases a very challenging task.

In this paper, we introduce the problem of *query cleaning* for keyword search queries in a database context and propose a set of effective and efficient solutions. Query cleaning involves semantic linkage and spelling corrections of database relevant query words, followed by segmentation of nearby query words such that each segment corresponds to a high quality data term. We define a quality metric of a keyword query, and propose a number of algorithms for cleaning keyword queries optimally. It is demonstrated that the basic optimal query cleaning problem can be solved using a dynamic programming algorithm. We further extend the basic algorithm to address incremental query cleaning and top-k optimal query cleaning. The incremental query cleaning is efficient and memory-bounded, hence is ideal for scenarios in which the keywords are streamed. The top-k query cleaning algorithm is guaranteed to return the best k cleaned keyword queries in ranked order. Extensive experiments are conducted on three real-life data sets, and the results confirm the effectiveness and efficiency of the proposed solutions.

1. INTRODUCTION

Keyword search has recented a great deal of attention both from researchers and practitioners. Popularized by World Wide Web (WWW) search engines, keyword search is becoming a common way for users to access data repositories such as XML documents and relational data warehouses. Recent research on keyword search of relational databases has revealed that the search space for answers of keyword queries of relational databases is much larger than that of the traditional information retrieval (IR) case. The reason for the exponential explosion of the search space is due to the additional task of assembling tuples from different tables into a complete view (in the form of a join network of tuples [8, 6] or a database view [17]) which contains all (or as many as possible) of the user specified keywords.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

1.1 Motivation: dirty queries

The difficulty of keyword search in databases is further exacerbated when the keyword query is *dirty*, i.e., it is contaminated by words which are not intended as part of the query. Another type of *dirty* words is misspelled words or words which do not appear in the database, but is semantically equivalent to some words in the database. Here are some sample scenarios in which dirty keywords occur.

Scenario 1: The keyword query is specified as a natural language sentence, hence contains words which are not database related.
Example: The user may specify the query as:
"What is the beer bought by Allison Ross?"

The only database relevant keywords are "beer" (product name), "bought" (relation name) and "Allison Ross" (customer). The rest "What is the $\cdots$ by $\cdots$" are part of the natural language. Traditional approach is to filter them out as *stop words*, but these words can potentially be database relevant in other context. For instance, the product name `Microsoft IS` contains the word "is" as an abbreviation for Information Server.

Scenario 2: The keywords are misspelled unintentionally, or users who are not familiar with the content of the database may use semantically equivalent or similar words which do not appear in the database at all.
Example: The user may misspell the name *Allison Ross* as "Alison Rose". Therefore, a query such as "Alison Rose water" may in fact be a dirty version of the true keywords "Allison Ross, Water", or "Allison, Rose Water".

Scenario 3: The keyword query is not user specified, but rather is embedded in a body of text, such as an e-mail, blogs or text messaging. Example: Matching bodies of text with advertising postings is a common technique used in internet marketing (e.g. Google's AdSense, or Google Mail's Ad posting). An exciting possibility is to extract a high quality keyword query from e-mails for the purpose of database search. Given an original body of text below, only certain words (underlined) are relevant to the database.
"Hi Allison,"
"Please let me know the beer that you bought."
"Also what is the best city to purchase ice wine?"

The difficulty is not only to select the most database relevant words from the body of text, but also to group words into multiple queries by taking into account their positions in the text.

Aside from dirty words, another potential problem is that keyword queries are normally a sequence of words separated by whitespaces, yet entries of databases are typically short sequences of words. Ideally, neighbouring query words are segmented into segments which are matched against database tuples. Traditional IR keyword search techniques matches the query against documents, and do not work well when the search result requires assembling multiple tu-

ples together to cover all the query words.

1.2 Keyword query cleaning

We propose to introduce a preprocessing stage to clean the raw text and extract high quality keyword queries. The added query cleaning will not only improve the quality of the search result, but will also significantly reduce the search space for the subsequent search algorithm. This enables one to apply search algorithms which are only suitable for queries with few keywords to process medium to large bodies of text such as blogs and emails. We argue that the reduction of the search space is significant enough that the improvement of the search runtime (e.g. time it takes to assemble join networks [8]) will well justify the overhead incurred by the additional cleaning phase.

The query cleaning algorithm must be able to filter out database irrelevant words, and identify misspelled words, synonyms and semantically equivalent words. The cleaning algorithm should also perform segmentation. When appropriate, neighbour words should be grouped together into segments if there is strong support that for considering the words as a single multi-word query term. Segmentation should be order insentative, i.e. "Allison Ross" and "Ross, Allison" should be both be grouped into a single query term. Finally, when extracting keyword queries from bodies of text, the query cleaning algorithm should take into considerations the relative positions of words so words that are far apart are less likely to be grouped together.

In order to address the issue of ambiguity of how a query should be cleaned, we introduce a scoring model to assign a score to a cleaned query, and define the top-k optimal query cleaning problem. This allows subsequent search algorithms to optimize search strategy for multiple possible clean keyword queries.

In many scenarios, the dirty keyword queries are obtained in a streamed fashion. We would like to handle dirty keyword streams using incremental query cleaning. For instance, if one is to offer search capability to text chat sessions, then it is important for the query cleaning algorithm to perform segmentation on the existing text, and incrementally improve the intermediate segmentation when new query tokens arrive. In the case of user-interactive search (similar to Google Suggest[1]), the query cleaning algorithm must produce intermediate segmentations responsively and incremental improvements as the user enters more keywords. Another scenario for streaming query cleaning is that, as part of a complete keyword query process, it is important for the query cleaning phase to be non-blocking so that the overall query processing can be pipelined. In order to handle long lasting keyword streams, streaming query cleaning needs to possess the following properties:

- Intermediate segmentations can be incrementally modified and improved *efficiently*.
- The incremental adjustment of intermediate segmentations exhibits expected constant run-time and memory with respect to the growing query length, so, the query cleaning algorithm does not require more time nor memory for incremental cleaning.

1.3 Contributions

We have made a number of contributions to the problem of keyword query cleaning.

- We formulate the keyword query cleaning problem a combinatoric search problem. The search space consists of all possible *segmentations* and modifications of query tokens. Our framework takes into account of semantic synonym expansion, spelling error correction, token permutation, and database term grouping. Each segmentation corresponds to a candidate keyword query. The quality of the cleaned query is formalized by a cost model which assigns scores to segmentations.
- We show that optimal query cleaning is NP-hard in general, but solvable in polynomial time if the *database term* length is bounded. We construct an optimal query cleaning algorithm based on dynamic programming.
- We construct an efficient top-k version of the query cleaning algorithm to improve the recall factor by applying the Fagin's algorithm in the dynamic program.
- We show that the optimal query cleaning for database with bounded term length can be solved incrementally. Namely, a dirty query that is streamed one keyword at a time can be cleaned more efficiently by an incremental query cleaning algorithm. More importantly, the incremental query cleaning algorithm we construct has a constant *expected* run-time and memory requirement, making it suitable for streaming keyword cleaning.

1.4 Outline of the paper

In Section 2, we discuss the existing literature on keyword queries for databases. We will also discuss relevant techniques from fields of natural language processing, computational linguistics and information retrieval. In Section 3, we formally define the problem of query cleaning as a cost-based optimization problem. The cost model takes into account of various noise discussed above. Section 4 to Section 6 present the algorithmic solutions for the basic optimal segmentation problem and its extensions (incremental and top-k segmentation). In Section 7, we will outline how query cleaning can benefit existing search algorithms in terms of both accuracy and performance. The algorithms are thoroughly evaluated in Section 8. We evaluated the algorithms against several distinctly different databases, and measured the performance in terms of computation time and accuracy. Section 9 concludes the paper with a summary and outlines the future work.

2. RELATED WORK

There has been a great deal of recent work in keyword queries for database systems. Early work [1, 8, 6] on keyword search queries for relational databases uses classical IR scores to find ways to join tuples from different tables. The search algorithms focus on enumeration of *join networks* to connect relevant tuples by joining different relational tables. The optimal join network problem has been shown to be NP-complete w.r.t. the number of relevant tables [8, 3], and heuristic algorithms for top-k enumeration of candidate networks have been proposed (e.g. [3, 6]). Recent work by Luo et. al. [10] and Liu et. al. [9] on relational database search focuses on more meaningful scoring functions and generation of top-k join networks of tuples. Markowetz et. al. [12] addresses the issue of keyword search on streams of relational data. Wu et. al. [17] introduce keyword search for relational databases with star-schemas found in OLAP applications. Keyword search for other data models, such as XML [5, 7], has also been studied.

Our work is complementary to the search algorithms proposed so far, in that the proposed query cleaning phase produces a keyword query from user input that can be better evaluated. Fuzzy keyword evaluations such as spelling correction and semantic matching are not dealt with by existing database search algorithms, but

[1] `http://labs.google.com/suggestfaq.html`

with query cleaning, the user input query is rewritten to a more database relevant query by token expansion (Section 3). The clean queries will be also evaluated with greater efficiency because query cleaning reduces the search space, in terms of relevant tables and tuples to be joined, by reducing the length of the query. This is done by means of segmentation (Section 3).

The core of keyword query cleaning is the problem of *segmentation*. Segmentation of words has been studied extensively in the literature of computational linguistics [15] and information extraction [11, 14]. Existing segmentation algorithms are based on training a probabilistic model of the language based on an existing corpus and maximal likelihood estimation of the positions of delimiters. These probabilistic segmentation algorithms do not apply to our segmentation problem for several reasons. First, because they are not intended for search, it is not guaranteed that each segment is actually part of the corpus, so it is possible that the segmented query term is not even in the database. Another problem with these segmentation algorithms is that they strictly enforce the sequential ordering of words, but flexible information retrieval algorithms support order insensitive search. Finally, it is not straight-forward to enable spelling correction and semantic matching in these algorithms.

3. QUERY CLEANING: PROBLEM FORMULATION

In this section, we formally define the problem of query cleaning.

DEFINITION 1 (TOKENS AND TERMS). Tokens *are strings which are considered as indivisible units. A* term *is a sequence of tokens. Let D be a database (relational or XML). A database token is a token which appears in somewhere in the database. The set of all database tokens of D is denoted by* TOKEN^D. *Similarly, we define a database term to be a term which appears in the database, and denote all the database terms as* TERM^D.

Note, we always assume that terms are *short* sequences of tokens. For large text values stored in the database, we may break them into short terms as is done in information retrieval [2].

DEFINITION 2 (INPUT QUERY). *An input query Q is a pair (t_Q, p_Q) where $t_Q = \langle t_Q(1), t_Q(2), \ldots, t_Q(n)\rangle$ is a sequence of tokens, and $p_Q = \langle p_Q(1), p_Q(2), \ldots, p_Q(n)\rangle$ is a sequence of monotonically increasing integers. The value $p_Q(i)$ is the position of the token $t_Q(i)$ in the query Q. The number of tokens in Q is its length $|Q|$.*

EXAMPLE 1. *Consider the query:*
Q ="harrison ford and directed by steven spielberg". The tokens and position values are as follows.

$t_Q(i)$	*harrison*	*ford*	*and*	*directed*	*by*	*steven*	*spielberg*
$p_Q(i)$	*1*	*2*	*3*	*4*	*5*	*6*	*7*

The positions $p_Q(i)$ do not have to be continuous. If one applies stop-words filter to the query, then the token "and", "by" will be eliminated. Yet we still keep the absolute positions of the pre-filtered query, so Q becomes:

$t_Q(i)$	*harrison*	*ford*	*directed*	*steven*	*spielberg*
$p_Q(i)$	*1*	*2*	*4*	*6*	*7*

DEFINITION 3 (TOKEN EXPANSION). *An expansion (with expansion factor m) is a function which maps tokens to a collection of m scored database tokens.*

$$\begin{aligned} \mathit{expand}_m &: \text{TOKEN} \rightarrow \mathit{list}\left\langle \text{TOKEN} \times \mathbb{R}^+ \right\rangle \\ &: t \mapsto \left\langle (t'_1, d_1), (t'_2, d_2), \ldots (t'_m, d_m) \right\rangle \end{aligned}$$

where for all $1 \leq i \leq m$, $t'_i \in \text{TOKEN}^D$.

The expansion function expand_m models the matching between query tokens t_Q and tokens in TOKEN^D by (1) correction of spelling errors, and (2) translation to semantically similar synonyms. The expanded tokens $\{t'_i\}_{i \leq m}$ are the top-m database tokens, and the distance between the query token t and t'_i is given by d_i. For spelling correction, the distance can be variants of string edit distances or cosine similarity of the q-grams between t and t'_i. The expansion function may easily be implemented using spell checking against the corpus of known database tokens or word associate using semantic lexical databases such as WordNet [13].

EXAMPLE 2. *Consider the query "Gerge Michael Jacksons Fife". Suppose that the database consists of musician and album names, and only spelling corrections are made by the expansion function with expansion factor $m = 3$. Then the expansion of "Gerge" is given by:*

$$\begin{aligned} & \mathit{expand}_3(\text{“}Gerge\text{”}) \\ = \; & \langle(\text{“}George\text{”}, 1), (\text{“}Gerbo\text{”}, 2), (\text{“}Georgia\text{”}, 3)\rangle \end{aligned}$$

In this example, we assumed that the distances d_i are simply the string edit distance.

DEFINITION 4 (EXPANSION MATRIX). *Given Q of length n and an expansion function expand with expansion factor m. An* expansion matrix *M is a $m \times n$ matrix of scored tokens obtained by placing the token expansion $\text{expand}_m(t_Q(i))$ as the i-th column of the matrix.*

$$M = \begin{bmatrix} (t_{11}, d_{11}) & (t_{12}, d_{12}) & \cdots & (t_{1n}, d_{1n}) \\ (t_{21}, d_{21}) & (t_{22}, d_{22}) & \cdots & (t_{2n}, d_{2n}) \\ \vdots & \vdots & \vdots & \vdots \\ (t_{m1}, d_{m1}) & (t_{m2}, d_{m2}) & \cdots & (t_{mn}, d_{mn}) \end{bmatrix}$$

where $\text{expand}_m(t_Q(i)) = \langle(t_{1i}, d_{1i}), \ldots, (t_{mi}, d_{mi})\rangle$.

We denote token t_{ij} of an expansion matrix M as $t_M(i, j)$, and its distance measure d_{ij} as $d_M(i, j)$.

EXAMPLE 3. *Continuing with Example 2, we can construct the expansion matrix for query Q, and the expansion function expand_3:*

$$M = \begin{bmatrix} (\text{“}George\text{”}, 1) & (\text{“}Michael\text{”}, 0) & (\text{“}Jacksons\text{”}, 0) & (\text{“}Five\text{”}, 1) \\ (\text{“}Gerbo\text{”}, 2) & (\text{“}Michaels\text{”}, 1) & (\text{“}Jackson\text{”}, 1) & (\text{“}Fifo\text{”}, 1) \\ (\text{“}Georgia\text{”}, 3) & (\text{“}Michigan\text{”}, 4) & (\text{“}Jacko\text{”}, 2) & (\text{“}Fifth\text{”}, 2) \end{bmatrix}$$

DEFINITION 5 (SEGMENTS AND SEGMENTATION). *Given an expansion matrix M of dimension $m \times n$, a* segment *is a sequence of entries in M, i.e.*

$$S = \langle(i_1, j_1), (i_2, j_1 + 1), \ldots (i_L, j_1 + L)\rangle$$

where $(i, j) \in S$ correspond to the entry $M(i, j)$. Let $\text{start}(S)$ *and* $\text{end}(S)$ *denote the first column j_1 and last column $j_1 + L$ covered by S respectively. The term induced by the segmentation S is defined as $\langle t_M(i, j) : (i, j) \in S\rangle$, denoted by T_S.*

A segmentation *is a sequence of segments $\mathbb{S} = \langle S_1, S_2, \ldots, S_K\rangle$ where for all $k \leq K$,* $\text{end}(S_k) + 1 = \text{start}(S_{k+1})$. *Namely, the segments are continuous and non-overlapping. We define the start and end of a segmentation as* $\text{start}(\mathbb{S}) = \text{start}(S_1)$ *and* $\text{end}(\mathbb{S}) = \text{end}(S_K)$. *A segmentation is* complete *if* $\text{start}(\mathbb{S}) = 1, \text{end}(\mathbb{S}) = n$, *and* partial *otherwise. A sub-segmentation $\mathit{subSeg}(\mathbb{S}, i, j)$ is defined as*

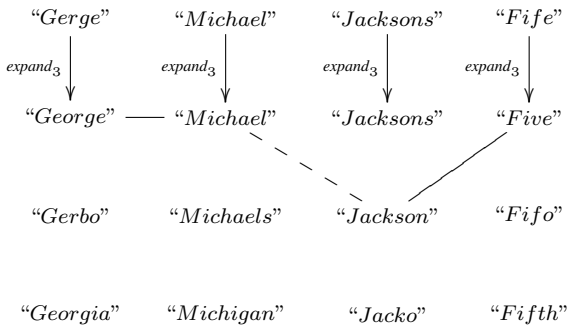
$$\mathit{subSeg}(\mathbb{S}, i, j) = \langle S \in \mathbb{S} : \text{start}(S) \geq i \text{ and } \text{end}(S) \leq j\rangle$$

Intuitively, a segment associates multiple *expanded* tokens together to form a query term, thus a segmentation groups tokens into a new query of multiple terms. We will see that given a scoring function for segmentations, the optimal segmentation $\mathbb{S}^*$ corresponds to the cleaned version of the original query Q.

EXAMPLE 4. *Continue with the previous example, segment $S_1 = \langle (1,1),(1,2)\rangle$ corresponds to the term "George Michael", and segment $S_2 = \langle (1,2),(2,3)\rangle$ corrseponds to the term "Michael Jackson". However $\langle S_1, S_2\rangle$ is not a valid segmentation because the overlap. There are many complete segmentations w.r.t. the matrix in Example 2. Consider the following two segmentations:*

$$\begin{aligned}\mathbb{S}_1 &= \left\langle \langle (1,1),(1,2)\rangle, \langle (2,3),(1,4)\rangle \right\rangle \\ \mathbb{S}_2 &= \left\langle \langle (1,1)\rangle, \langle (1,2),(2,3)\rangle, \langle (1,4)\rangle \right\rangle\end{aligned}$$

They are shown as edges ($\mathbb{S}_1$ is solid, and $\mathbb{S}_2$ dashed) below.



The two segmentations are two different ways of interpreting the query Q. The first segmentation produces the query "(George Michael), (Jackson Five)"" while the second segmentation produces the query "(George), (Michael Jackson), (Five)". Both segmentations are potential intentions of the user, but one would agree that $\mathbb{S}_1$ is a more sensible guess than $\mathbb{S}_2$. Both segmentations "fixed" the spelling errors in the original query Q.

As the example demonstrates, there are multiple "sensible" segmentations. In order to distinguish the quality of different segmentations, we formally define a scoring function. The scoring function is simply to calculate, and we believe coincides well with user intuition. A score is assigned to each segment in the segmentation, and these scores are aggregated by summation to form the score of the entire segmentation.

DEFINITION 6 (SEGMENTATION SCORE). *Given Q and its expansion matrix M. The score of a single segment S is defined in terms of:*

- *max query distance:*

  $$\delta_Q(S) = \max\{p_Q(i+1) - p_Q(i) - 1 : \text{start}(S) \le i < \text{end}(S)\}$$

- *total expansion distance*

  $$\delta_{exp}(S) = \sum_{(i,j)\in S} d_M(i,j)$$

- *information retrieval score of the segment induced terms:*

  $$\text{SCORE}_{\text{IR}}(T_S) = \max\{\textbf{\textit{tfidf}}(T_S, T) : T \in \text{TERM}^D\}$$

  *where **tfidf** is the classical tf-idf weight between the term T_S and the database term T. We treat the data term T as a document, and the query term T_S as a single term. The entire database terms form the document collection.*

Let $\text{NORMALIZE} : \mathbb{R}^+ \times \mathbb{R}^+ \to [0,1]$ *be a normalization function which is anti-monotonic, and*

$$\lim_{x\to\infty, y\to\infty} \text{NORMALIZE}(x,y) = 0$$
$$\text{NORMALIZE}(0,0) = 1$$

Let a boost function $\text{BOOST} : \mathbb{N} \to \mathbb{R}^+$ *is a monotonic function with* $\text{BOOST}(n) \ge 1$. *The final score is given by*

$$\begin{aligned}&\text{SCORE}(S) \\ =\ &\text{SCORE}_{IR}(T_S) \cdot \text{NORMALIZE}(\delta_Q(S), \delta_{exp}(S)) \cdot \text{BOOST}(|T_S|)\end{aligned}$$

The score of a segmentation is the sum of all the scores of its segments:

$$\text{SCORE}(\mathbb{S}) = \sum_{S\in\mathbb{S}} \text{SCORE}(S)$$

The rationale behind Definition 6 is the following.

(1) We prefer if tokens that are grouped into a single segment are adjacent to each other in the user specified query. Hence, $\text{SCORE}(S)$ favours S with smaller values of $\delta_Q(S)$.

(2) We also prefer to minimize the changes made to the original tokens in Q, hence we favour smaller values of $\delta_{\exp}(S)$.

(3) If a long sequence of segments is found, then this is a valuable finding, and we would like to boost its score by its length $|T_S|$ according to the function $\text{BOOST}()$.

Observe that those objective functions are potentially conflicting, i.e., longer segmentations may increase $\delta_Q(S)$ (unfavoured by rational (1)) but enjoys a higher boost (favoured by rational (3)). We combine these multiple objectives into a single score multiplicatively.

DEFINITION 7 (OPTIMAL SEGMENTATION). *Given an expansion matrix M of a query, the optimal segmentation $\mathbb{S}^*$ is one that is a complete segmentation and maximizes the score* $\text{SCORE}(\mathbb{S}^*)$. *The top-$k$ segmentations of M are the k complete segmentations with the k highest scores.*

Algorithms which compute the optimal segmentation and top-k segmentations serves as the basis of the solution to the query cleaning problem.

4. OPTIMAL SEGMENTATION

In this section, we present (in)tractability results and algorithms for the optimal segmention problem. Algorithms described in this section will form the basis of incremental and top-k computation of segmentations.

4.1 Dynamic programming

The optimal segmentation problem can be solved by dynamic programming.

Given an expansion matrix M with n columns, let $M(i)$ be the i-th column vector in M, and

$$M(i\ldots j) = [M(i)M(i+1)\ldots M(j)]$$

Therefore, $M(1\ldots n) = M$. Denote $\mathbb{S}^*_{i,j}$ to be the optimal segmentation of the sub-matrix $M(i\ldots j)$. Let $\mathbb{S}^* = \mathbb{S}^*_{1,n}$ Finally, we also define the optimal segment $S^*_{i,j}$ which is a single segment covering columns $i, i+1, \ldots, j$. A recursive relation can be established.

THEOREM 1 (RECURSIVE COMPUTATION OF $\mathbb{S}^*$). *Define a set of segmentations $\mathcal{X}_{i,j}$ as:*

$$\mathcal{X}_{i,j} = \{\mathbb{S}^*_{i,k} \oplus \mathbb{S}^*_{k+1,j} : i \le k < j\} \cup \{S^*_{i,j}\}$$

where $\oplus$ catenates two segmentations. The optimal segmentation of $M(i \dots j)$ is given by:

$$\mathbb{S}^*_{i,j} = \text{ARGMAX}\{\text{SCORE}(\mathbb{S}) : \mathbb{S} \in \mathcal{X}_{i,j}\}$$

Namely, $\mathbb{S}^*_{i,j}$ is the segmentation in $\mathcal{X}_{i,j}$ with the highest score.

Theorem 1 provides the basis of a dynamic programming solution which needs to compute only at most n^2 sub-problems. However, part of the dynamic program is to solve for the optimal segment $S^*_{i,j}$ spanning columns i to j inclusively. This, unfortunately, is intractable in general.

THEOREM 2 (INTRACTABILITY OF S^*). *Given an expansion matrix $M(i \dots j)$, computing the optimal segment $S^*_{i,j}$ is NP-hard with respect to the number of tokens, n.*

PROOF OUTLINE. We can reduce the optimal segment problem to the knapsack problem. Given an instance knapsack problem with n items, with the weights w_i and values v_i. Let the maximal capacity be c. Recall that the decision problem is whether there exists a subset of the n items such that the total value is greater than k, but with total weight less than c. We construct the matrix M as follows:

$$M = \begin{bmatrix} (t_1, 0) & (t_2, 0) & \dots & (t_n, 0) \\ (t'_1, 1) & (t'_2, 1) & \dots & (t'_n, 1) \end{bmatrix}$$

We construct a database D such that

$$\text{SCORE}_{\text{IR}}(T_S) = |\{t_i \in S\}|$$

This can always be done by designing the content of the database. Since we are focused on the query complexity instead of data complexity, we are not concerned about the size of the database. Let the normalization function be such that

$$\text{NORMALIZE}(\delta_Q(S), \delta_{\text{exp}}(S)) = \begin{cases} 1 & \text{if } \delta_{\text{exp}}(S) < c, \\ 0 & \text{else.} \end{cases}$$

Then, we can show that the optimal segment $\text{SCORE}(S^*) \ge k$ if and only if the corresponding knapsack problem has a solution. This concludes the reduction. □

In order to compute optimal segmentation using Theorem 1, we need an efficient algorithm for computing $S^*_{i,j}$. Despite the general intractability result in Theorem 2, we can still compute $S^*_{i,j}$ reasonably efficiently in most cases. In fact, it is easy to see that if the lengths of database terms are bounded, then the complexity of the optimal segmentation problem collapses to polynomial time. This is made precise by Theorem 3 as follows.

DEFINITION 8 (QUERY-INDUCED TERMS). *Given a query Q with an expansion matrix M, and a database D. Define the query-induced database terms $\text{TERM}^D(Q)$ as all the database terms that intersect with the expansion matrix:*

$$\text{TERM}^D(Q) = \{T \in \text{TERM}^D : T \text{ contains a token } t \text{ in } t_M\}$$

THEOREM 3. *Suppose that all terms $T \in \text{TERM}^D(Q)$ are such that $|T| \le k$, then $S^*_{i,j}$ can be computed in $\mathcal{O}(n^k)$ for all $1 \le i \le j \le n$.*

The algorithm shown in Algorithm 1 performs greedy path scan in the sub-matrix $M(i \dots j)$ by keeping only the best L paths. If the paths cannot be extended further, then the search is terminated and returns that no single segment can span $M(i \dots j)$. The optimal segment is found by $S^*_{i,j} = \text{ARGMAX}\{\text{SCORE}(S) : S \in \text{TOPSEGMENTS}(M, i, j)\}$. In practice, the average time complexity of the computation of $S^*_{i,j}$ described in Section 4.2 is much better than even $\mathcal{O}(n^k)$ because for a typical database, for large enough sub-matrix $M(i \dots j)$, the procedure TOPSEGMENTS shown in Algorithm 1 very quickly deduces that there can be no *single* segment that covers columns from i to j.

4.2 The dynamic programming solution

Algorithm 1 TOPSEGMENTS(M, i, j): computes optimal segment to cover columns i to j in matrix M.

Require: an integer parameter $L > 0$.
1: **if** $i = j$ **then**
2: **return** the top-L tokens in $M(i)$ as segments
3: **else**
4: $A = \text{TOPSEGMENTS}(M, i, j-1)$
5: $B = \{S \oplus t : S \in A, t \in M(j), T_{S \oplus t} \in \text{TERM}^D\}$
6: **return** the top-L segments in B
7: **end if**

We know that if $L = n^k$ as defined in Theorem 3, then TOPSEGMENT guarantees to contain the global optimal. In practice, the cardinality of the candidate set B is quite small by the pruning condition ($T_{S \oplus t} \in \text{TERM}^D$) in Line 5 of Algorithm 1.

Algorithm 2 BOTTOMUPSEGS(M)

1: $n =$ number of columns of M.
2: *optSegs* = new matrix of size $n \times n$.
3: **for** $i = 1 \dots n$ **do**
4: *optSegs*$(i, i) = \langle S^*_{i,i} \rangle$
5: **end for**
6: **for** $c = 1 \to n - 1$ **do**
7: **for** $i = 2 \to n - c$ **do**
8: $j = i + c$
9: $A = \{$*optSegs*$(i, k) \oplus$ *optSegs*$(k + 1, j)$
10: $: 1 \le k < j\} \cup \{\langle S^*_{i,j} \rangle\}$
11: *optSegs*$(i, j) = \text{ARGMAX}\{\text{SCORE}(\mathbb{S}) : \mathbb{S} \in A\}$
12: **end for**
13: **end for**
14: $\mathbb{S}^* =$ *optSegs*$(1, n)$
15: **return** (*optSegs*, $\mathbb{S}^*$)

One may verify that *optSegs*$(i, j) = \mathbb{S}^*_{i,j}$. It is straight forward to check that the time complexity of algorithm BOTTOMUPSEGS in Algorithm 2 is $\mathcal{O}(n^3 \cdot \mathbf{T}(\text{TOPSEGMENTS}))$, where $\mathbf{T}(\text{TOPSEGMENTS})$ is the time complexity of TOPSEGMENTS. Therefore, by Theorem 3, the overhaul computation of the optimal segmentation can be done in $\mathcal{O}(n^c)$ for some $c > 3$. If the bound is too large, one may apply the heuristics of separating long database text into smaller pieces, thus ensuring that the length of each database token in the index is always bounded. Note that the BOTTOMUPSEGS will compute the optimal segmentation for any scoring function satisfying Theorem 3.

4.3 Scoring functions

The segmentation scoring function is completely characterized by the normalization function NORMALIZE and the boosting func-

tion BOOST. These functions are used to penalize large query distances and semantic distances, and favor long segments. We choose to use the following simple formuli for these functions.

$$\begin{aligned} \text{NORMALIZE}(x,y) &= e^{-(\alpha x+\beta y)} \text{ where } \alpha,\beta>0, \\ \text{BOOST}(n) &= (1+\gamma)n \quad \text{where } \gamma>0 \end{aligned}$$

The tunable parameters α, β and γ reflect the sensitivity to query distance, spelling errors / semantic differences, and long segments in the query cleaning phase. We will evaluate the impact of different parameter choices in the experimental section.

EXAMPLE 5. *Consider the two segmentations $\mathbb{S}_1$ and $\mathbb{S}_2$ in Example 2. Segmentation $\mathbb{S}_1 = \langle S_1, S_2\rangle$ corresponds to the keyword query of "(George Michael), (Jackson Five)". The first segment $S_1 = \langle 1,2\rangle$ has a query distance $\delta_Q(S_1) = 0$ because the tokens "George" and "Michael" are without a gap in the original query. Similarly the second segment S_2 is also with $\delta_Q(S_2) = 0$. In terms of expansion distances, $\delta_{\exp}(S_1) = 1$ because "George" in the first segment S_1 was not the original token, but rather an expanded token with a distance[2] of 1. Similarly, $\delta_{\exp}(S_2) = 2$ because "Jackson" in the second segment S_2 was an expanded token with a distance of 1 to the original token, and "Five" is also 1 distance away from the original token "Fife". For simplicity, let us assume the values $\alpha = \beta = 1$. Therefore,*

$$\begin{aligned} \text{SCORE}(S_1) &= (2\gamma)\cdot \text{SCORE}_{\text{IR}}(\text{“}George\ Michael\text{”}) \\ \text{SCORE}(S_2) &= \frac{2\gamma}{e^2}\cdot \text{SCORE}_{\text{IR}}(\text{“}Jackson\ Five\text{”}) \end{aligned}$$

The total score of the segmentation
$\mathbb{S}_1 = \text{SCORE}(S_1) + \text{SCORE}(S_2)$.

The second segmentation $\mathbb{S}_2 = \langle S_1, S_2, S_3\rangle$ in Example 2 has three segments corresponding to terms "(George), (Michael Jackson), (Five)". One can easily verify that their respective scores are:

$$\begin{aligned} \text{SCORE}(S_1) &= \gamma\cdot \text{SCORE}_{\text{IR}}(\text{“}George\text{”}) \\ \text{SCORE}(S_2) &= \frac{2\gamma}{e^2}\cdot \text{SCORE}_{\text{IR}}(\text{“}Michael\ Jackson\text{”}) \\ \text{SCORE}(S_3) &= \frac{\gamma}{e}\cdot \text{SCORE}_{\text{IR}}(\text{“}Five\text{”}) \\ \text{SCORE}(\mathbb{S}_2) &= \text{SCORE}(S_1)+\text{SCORE}(S_2)+\text{SCORE}(S_3) \end{aligned}$$

The quality of segmentations $\mathbb{S}_1$ and $\mathbb{S}_2$ can be decided by comparing their scores. For simplicity, let us assume that the database content is such that $\text{SCORE}_{\text{IR}}(T) = c$ is a constant. Then, we have:

- *$\mathbb{S}_1$ is better than $\mathbb{S}_2$ if $\gamma > 0$, and*
- *$\mathbb{S}_1$ is equivalent to $\mathbb{S}_2$ if $\gamma = 0$.*

This agrees with the fact that γ controls the preference of longer segments. We find that values for $\gamma \in [0.5, 1]$ work quite well. Thus, a typically segmentation scoring function will prefer $\mathbb{S}_1$ over $\mathbb{S}_2$, as most readers would agree.

5. INCREMENTAL AND STREAMING SEGMENTATION

In this section, we consider the problem of incrementally computing the optimal segmentation when the user query is appended with new tokens. Given a query Q with $|Q| = n$, let us assume that the query is to be appended by an additional token to form a new query $Q' = \langle Q\ t'\rangle$. A naive recomputation of the optimal segmentation is clearly undesirable because it requires at least $\mathcal{O}(n^3)$.

[2]We assume a simple string edit distance measure in the example.

Algorithm 3 INCOPTSEGMENTS$(M, \textit{optSegs}, t')$

Require: *optSeg* is the matrix of optimal segmentation. {The new query $Q' = \langle Q\ t\rangle$}
1: $n =$ number of keywords in M.
2: reallocate *optSeg* to be $(n+1)\times(n+1)$
3: **for** $i = n+1 \to 1$ **do**
4: $\quad A = \{\textit{optSegs}(i,k) \oplus \textit{optSegs}(k+1, n+1) : i \le k < n+1\}$
$\qquad \cup \{S^*_{i,n+1}\}$
5: $\quad \textit{optSegs}(i, n+1) = \text{ARGMAX}\{\text{SCORE}(\mathbb{S}) : \mathbb{S} \in A\}$
$\quad$ {Check if we can return early.}
6: $\quad \mathbb{S}_{\textbf{LCP}} = \textbf{LCP}(\textit{optSegs}(i,n), \textit{optSegs}(i,n+1))$
7: $\quad j = \text{end}(\mathbb{S}_{\textbf{LCP}})$
8: $\quad$ **if** $\mathbb{S}_{\textbf{LCP}} \neq \langle\rangle$ **then**
9: $\quad\quad \forall i' \le i,\ \textit{optSegs}(i', n+1) =$
$\qquad\quad\quad \textit{subSeg}(\textit{optSegs}(i', n), i', j) \oplus \textit{optSegs}(j+1, n+1)$
10: $\quad\quad$ **return**
11: $\quad$ **end if**
12: **end for**

5.1 Incremental segmentation

We can dramatically improve the computation of the optimal segmentation of the new query Q' based on the segmentation results of the previous query Q. Let us denote $\mathbb{S}'^*$ and M' the optimal segmentation and expansion matrix of query Q'. Let *optSegs'* be the matrix of segmentation that would be computed by BOTTOMUPSEGS(M'), i.e., $\mathbb{S}'^* = \textit{optSegs}'(1, n+1)$. The objective of the incremental segmentation is to compute $(M', \textit{optSegs}', \mathbb{S}'^*)$ incrementally based on the already calculated $(M, \textit{optSegs}, \mathbb{S}^*)$.

It is easy to see that the expansion matrix M' simply contains one additional column consisting of the scored expanded tokens from the new query token t'. Since $\textit{optSegs}'(i,j)$ is the optimal segmentation of the sub-matrix $M'(i \ldots j)$, we have the following result:

$$\textit{optSegs}'(i,j) = \textit{optSegs}(i,j) \text{ for all } i, j \le n.$$

Thus, one only needs to compute $\textit{optSegs}'(i, n+1)$ where $1 \le i \le n+1$. The naive incremental algorithm would compute n partial segmentations

$$\textit{optSegs}'(n+1, n+1), \textit{optSegs}'(n, n+1), \ldots \textit{optSegs}'(1, n+1)$$

Each of these requires $\mathcal{O}(n)$. So, the incremental computation is with a complexity of $\mathcal{O}(n^2)$, an improvement from $\mathcal{O}(n^3)$. Despite the improvement from the overhaul approach, the naive incremental computation is still dependent on the previous query length n which is ever increasing in query cleaning scenarios for infinite streams of keywords.

We now describe a version the incremental segmentation which has a $\mathcal{O}(1)$ expected run-time and memory usage. It makes use of the following result.

DEFINITION 9 (COMMON PREFIX). *Given a (partial) segmentation $\mathbb{S}$ of some expansion matrix M, a prefix $\mathbb{S}_1$ of $\mathbb{S}$ is some segmentation such that $\exists \mathbb{S}_2,\ \mathbb{S}_1 \oplus \mathbb{S}_2 = \mathbb{S}$, where $\oplus$ catenates two segmentations. Let $\mathbb{S}_1$ and $\mathbb{S}_2$ be two partial segmentations of some expansion matrix M. The longest common prefix of $\mathbb{S}_1$, $\mathbb{S}_2$ is denoted by $\textbf{LCP}(\mathbb{S}_1, \mathbb{S}_2)$.*

LEMMA 1 (LCP BOUNDARY).

$$\begin{aligned} & \textbf{LCP}(\mathbb{S}^*_{i,n}, \mathbb{S}'^*_{i,n+1}) \neq \emptyset \\ \Longrightarrow\ & \text{end}(\textbf{LCP}(\mathbb{S}^*_{i-1,n}, \mathbb{S}'^*_{i-1,n+1})) \\ & \quad = \text{end}(\textbf{LCP}(\mathbb{S}^*_{i',n}, \mathbb{S}'^*_{i',n+1})). \end{aligned}$$

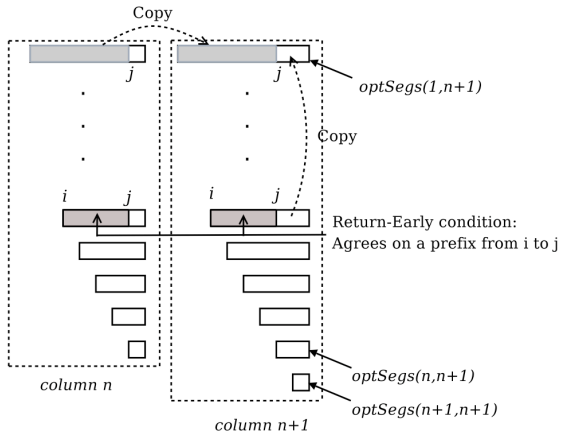


Figure 1: Return-Early condition and incrementation computation of *optSegs* matrix.

PROOF OUTLINE. Define $\mathbb{S}^*_{i,j}$ as the optimal segmentation for the sub-query from i to j for the query string with n tokens, and $\mathbb{S}'^*_{i,j}$ the optimal segmentation from i to j of the extended query string with $n+1$ tokens. We need to prove that if $\textbf{LCP}(\mathbb{S}^*_{i,n}, \mathbb{S}'^*_{i,n+1})$ spans over i to j for some $j \leq n$, then $\textbf{LCP}(\mathbb{S}^*_{i-1,n}, \mathbb{S}'^*_{i-1,n+1})$ spans over $i-1$ to j.

By hypothesis, j is a segment boundary in both $\mathbb{S}^*_{i,n}$ and $\mathbb{S}'^*_{i,n}$, hence is a segment boundary in $\mathbb{S}^*_{i-1,n}$ and $\mathbb{S}'^*_{i-1,n}$. Thus, we can write $\mathbb{S}^*_{i-1,n} = A_1 \oplus B_1$ and $\mathbb{S}'^*_{i-1,n} = A_2 \oplus B_2$ where A_i, B_i are segmentations and $\oplus$ is the catenation operator. One can show that our formulation of the $score()$ function is monotonic w.r.t. the segmentation catenation $\oplus$ operator. By the monotonicity w.r.t $\oplus$, $score(A_1) \geq score(A_2)$, and $score(A_1 + B_2) \geq score(A_2 + B_2)$. By the optimality of $\mathbb{S}'^*_{i-1,n+1}$, we have

$$\text{SCORE}(A_2 \oplus B_2) \geq \text{SCORE}(A_1 \oplus B_2) \implies \text{SCORE}(A_2) \geq \text{SCORE}(A_1)$$

Thus, $\text{SCORE}(A_1) = \text{SCORE}(A_2)$. This proves that A_2 is also optimal. Therefore $\textbf{LCP}(\mathbb{S}^*_{i-1,n}, \mathbb{S}'^*_{i-1,n+1})$ spans over $i-1$ to j. □

Lemma 1 states that if $\mathbb{S}^*_{i,n}$ and $\mathbb{S}'^*_{i,n+1}$ agree on segments from i to j, where $j = \text{end}(\textbf{LCP}(\mathbb{S}^*_{i,n}, \mathbb{S}'^*_{i,n+1}))$, then $\mathbb{S}^*_{i-1,n}$ and $\mathbb{S}^*_{i-1,n+1}$ will also agree on the segments from $i-1$ to j. By induction, we can further conclude that $\mathbb{S}^*_{1,n}$ and $\mathbb{S}'^*_{1,n+1}$ must also agree on the initial segments from 1 to j as illustrated in Figure 1. This allows us to more efficiently calculate the new segmentation $\mathbb{S}'^*$.

Algorithm 3 is a significant improvement from the naive incremental approach because as it is computing the new entries $optSegs(n+1, n+1), optSegs(n, n+1), \ldots, optSegs(1, n+1)$, it decides whether it can apply Lemma 1, and if so, it immediately fills up the subsequent entries in the $n+1$-column of *optSegs* matrix using entries in the n-column. We shall refer to the condition in Line 8 in Algorithm 3 as the *return-early* condition. In our implementation, this can be done using direct memory copy. However, in general, there is no guarantee when Lemma 1 applies, and in the worst case, Algorithm 3 may compute all the new entries from $optSegs(n+1, n+1)$ to $optSegs(1, n+1)$ with run-time in $\mathcal{O}(n)$.

5.2 Stream segmentation

We now show that in most cases, Algorithm 3 will return early in Line 10 after computing a constant number of entries. This important fact allows us to devise a constant-time and bounded-memory algorithm for performing query cleaning by segmentation for streams of keywords.

DEFINITION 10 (DEGREE OF CONNECTIVITY). *Two terms T_1, T_2 are* connected *if there exists a token t that appears in both T_1 and T_2. Given a set of terms $\mathcal{T}$, the connectivity graph is an undirected graph $G(\mathcal{T}) = (V, E)$ where $V = \mathcal{T}$ and $E = \{\{T_1, T_2\} : T_1, T_2 \text{ are connected}\}$. The degree of connectivity of the set of terms $\mathcal{T}$ is defined as the length of the longest cycle-free path in $G(\mathcal{T})$.*

LEMMA 2 (BOUNDED INCREMENTAL BACKTRACKING). *Given a query Q, and its expansion matrix M. Let c be the degree of connectivity of $\text{TERM}^D(Q)$ (as defined in Defintion 8). Then, we have* $\text{end}(\textbf{LCP}(\mathbb{S}^*_{1,n}, \mathbb{S}^*_{1,n+1})) \geq n + 1 - c$.

PROOF OUTLINE. Let c be the degree of connectivity. Given an expansion sub-matrix M with n columns where $n > c$. By the definition of degree of connectivity c, one can show that there must be a segmentation boundary k between $1, n$ such that terms in $\mathbb{S}^*_{1,k}$ do not share *any* tokens with terms in $\mathbb{S}^*_{k+1,n}$. Since $n - k < c$, we conclude $k > n - c$.

Let M' be the matrix obtained by extending M by one more column, and let $\mathbb{S}'^*$ be the optimal segmentation of M'. By the bounded degree of connectivity, we know that k is *still* a segmentation boundary for the extended matrix M'. Using a simple cut-and-past argument, we can conclude that $\mathbb{S}^*_{1,n}$ and $\mathbb{S}'^*_{1,n+1}$ agree up to k where $k > n - c$. □

Lemma 2 asserts that the return-early condition (Line 8. in INCOPTSEGMENTS is guaranteed to be satisfied after at most c iterations where c is the degree of connectivity of the query-induced database terms. Since the value of c is always bounded by the degree of connectivity of all the database terms, we can immediately conclude the following result.

COROLLARY 1. *Given that a database D is such that degree of connectivity of TERM^D is c_D, then the run-time of* INCOPTSEGMENTS *is in $\Theta(c_D^3) = \mathcal{O}(1)$.*

If we fix the database, then the run-time for incremental segmentation does not not depend on the query length! In order to accomodate infinite keyword streams, we must also provide a constant bound on the memory required. As it stands, INCOPTSEGMENTS extends the matrix *optSegs* by an additional column each time. In order to bound the memory consumption, we propose to devise a streaming incremental segmentation algorithm which converts a stream of keywords into a stream of segments. This is feasible due to the following result which follows immediately from Lemma 1 and Lemma 2.

THEOREM 4 (STATIONARY SEGMENTS). *Let $Q' = \langle Q\ t' \rangle$. Let n be the length of Q. Suppose $\mathbb{S}_{\textbf{LCP}} = \textbf{LCP}(\mathbb{S}^*, \mathbb{S}'^*) \neq \langle\rangle$, and let $j = \text{end}(\mathbb{S}_{\textbf{LCP}})$. Let c be the degree of connectivity of $\text{TERM}^D(Q')$. If $j < n - c$, then segments in $\mathbb{S}_{\textbf{LCP}}$ will be a prefix of $Q'' = \langle Q'\ t'' \rangle$ for any token t''.*

The importance of Theorem 4 is that it allows us to determine the segments which can *never* be modified by future incoming tokens in a stream. These *stationary* segments cover the initial columns of the expansion matrix M from 1 to j where j is the end of the common prefix given in Theorem 4. The streaming segmentation can safely place these segments in the output stream, and remove the columns $1 \ldots j$ from the expansion matrix, thus freeing the memory.

Algorithm 4 TOPK-SEGMENTS(M, i, j)

Require: M is a $m \times n$ expansion matrix, and $1 \leq i \leq j \leq n$.
{Returns k single segments which covers columns i to j with the top-k scores.}
return top-k segments from TOPSEGMENTS(M, i, j)

6. TOP-K SEGMENTATION

In Section 4 and Section 5, we have presented algorithms to compute the optimal segmentation of a given query. In order to accomodate the inherit imperfections in keyword search, we would like to relax the query cleaning algorithm to compute top-k segmentations of a given query. That is, the top-k segmentations are the k segmentations which have the k highest-scored segmentations of the query. In this section, we present an extension to the basic optimal segmentation algorithm (Algorithm 2) to compute the top-k segmentations in a bottom-up fashion. Our top-k algorithm performs *union* and *join* operations on top-k sub-segmentations and build up the final top-k list bottom up. We also outline how the top-k segmentation algorithm can be extended to perform incremental top-k segmentation.

6.1 Bottom-up top-k segmentations

DEFINITION 11 (UNION AND JOIN OF RANKED LISTS). *Let X be a set of objects with a scoring function* SCORE $: X \rightarrow \mathbb{R}^+$. *Given a ranked list of objects L, define* ARGMAX$_k\{$SCORE$(x) : x \in L\}$ *as the top-k objects in L.*

Let L_1 and L_2 be two ranked lists of objects. The top-k union $L_1 \cup_k L_2$ is defined as the ranked list containing the k objects from $L_1 \cup L_2$ with the highest scores. We may write it as

$$L_1 \cup_k L_2 = \text{ARGMAX}_k\{\text{SCORE}(x) : x \in L_1 \cup L_2\}$$

Let $f : X \times X \rightarrow X$ be a function on pairs of objects. The top-k f-join is defined as:

$$L_1 \bowtie_f L_2 = \text{ARGMAX}_K\{\text{SCORE} \circ f(x_1, x_2) : x_1 \in L_1, x_2 \in L_2\}$$

The *join* operator is a specialization of the join operator of data streams discussed by Fagin et. al. [4], thus it can be implemented using Fagin's algorithm.

LEMMA 3 (COMPUTING UNION AND JOIN). *Let L_1 and L_2 be two list of objects which are individually ranked already. Then the top-k union can be done with time complexity of $\mathcal{O}(k)$.*

If there exists a monotonic function $\hat{f} : \mathbb{R}^+ \times \mathbb{R}^+ \rightarrow \mathbb{R}^+$ such that for all $x, x' \in X$ such that

$$\text{SCORE} \circ f(x, x') = \hat{f}(\text{SCORE}(x), \text{SCORE}(x'))$$

then, the top-k join can be done with time complexity of $\mathcal{O}(k)$.

It is easy to see that the top-k union can be done in $\mathcal{O}(k)$ – we simply perform merging of the sorted lists L_1, L_2, and terminate when there are k elements merged. As for the top-k join, we can simply utilize Fagin's algorithm [4] by treating L_i as scores of all the objects which are of the same object label.

In our context, the objects are partial segmentations, and the join function f is the catenation function $\oplus$ of segmentations. By the definition of the scoring function defined in Section 3, it is easy to see that $\hat{f}$ is simply summation $+$.

Algorithm 5 computes the top-k segmentations. It computes a matrix of top-k sub-segmentations which are then combined in a bottom-up fashion using top-k join and union operations.

Algorithm 5 TOPK-SEGMENTATION(M)

Require: M is an expansion matrix of a query Q of size $m \times n$.
{ *topKSegs* is a matrix of size $n \times n$ in which each entry (i, j) is the top-k segmentations for the sub-matrxi $M(i \ldots j)$.}
1: **for** $d = 1 \rightarrow n - 1$ **do**
2: **for** $i = 1 \rightarrow n - d$ **do**
3: $j = i + d$
4: $A = \{topKSegs(i, k) \bowtie_k topKSegs(k + 1, n) : i \leq k < j\}$ $\cup$ TOPK-SEGMENTS(M, i, j)
5: $topKSegs(i, j) = \cup_k A$
6: **end for**
7: **end for**
8: **return** $topKSegs(1, n)$

6.2 Incremental top-k segmentation

Results on incremental computation of the optimal segmentation in Section 5 can easily be generalized to the case of top-k computation. Lemma 1 can be generalized to the longest common prefix of two top-k segmentations. Thus when computing the new $n + 1$-column of the matrix *topKSegs*, we can introduce a return-early condition that is analogous to Line 8 – Line 11 in Algorithm 3.

Results on bounded incremental backtracking (Lemma 2) and stationary segments (Theorem 4) also straight-forwardly generalize to top-k segmentation. Thus, stream segmentation with the bounded memory as outlined in Section 5.2 applies to top-k segmentation.

7. PRACTICAL IMPACTS OF QUERY CLEANING

In previous sections, we have presented token segmentation algorithms. By performing segmentation, we are able to perform spelling correction and semantic translation of query tokens, and group tokens into query terms. In this section, we discuss some practical issues of applying segmentation for query cleaning and its benefit to the subsequent keyword search algorithms.

Keyword filtering: We have assumed, so far, that all tokens in the query are relevant to the database. This is certainly not always the case, especially when we deal with keyword extraction from bodies of text messages. In order to perform segmentation only on database-relevant tokens, we perform keyword filtering using stop-words and *expansion score threshold*. The expansion score threshold is to impose a minimal threshold for the distance between all original query tokens to their closest database tokens. Given a query token t, recall the expansion function $\text{expand}_m(t) = \langle (t'_1, d_1), (t'_2, d_2), \ldots \rangle$ tries to match t to m most relevant database terms $t'_1, t'_2, \ldots$ with distance measures $d_1 < d_2 < \ldots$ respectively. If the smallest distance $d_1 > d^*$ where d^* is a specified threshold, then one may safely eliminate the query token t from segmentation.

From segmentations to keyword queries: Once we have performed the keyword filtering as described above, we keep the remaining query tokens *and* their *original* query position. Therefore, even when two tokens, $t_Q(i)$ and $t_Q(i + 1)$, are adjacent to each other in Q *after* keyword filtering, their positions $p_Q(i)$ and $p_Q(i + 1)$ may differ by arbitrary amount because there may have been many tokens in between that were filtered. We may also introduce additional gaps between $p_Q(i)$ and $p_Q(i + 1)$ if they were delimited by special puntuations such as comma, period or a paragraph separation. The extra query distance $p_Q(i + 1) - p_Q(i)$ decreases the likelihood of having $t_Q(i)$ and $t_Q(i + 1)$ grouped into a single segment. Keyword queries are formulated directly from

the segmentations computed by the algorithms described in Section 4–Section 6. Each multi-token segment forms a single query term, thus query cleaning significantly reduces the length of the final query.

8. EXPERIMENTS

8.1 Implementation and experiment setting

In order to evaluate the effectiveness and efficiency of the proposed algorithms, we prototyped them within a system designed to support keyword queries in databases, which is under development at UOIT and York University. Extensive experiments were conducted on three real-life data sets under a variety of parameter settings.

The data sets we use are the Internet Movie Database (IMDB)[3], the DBLP data set[4], and the FoodMart sample database shipped with Microsoft SQL Server 2005. The first two data sets are obtained and processed in exactly the same way as what is done in [10]. The IMDB data contains information on movies, actors, directors, and so on, and it has 9,839,026 tuples. The raw text files in this data set are converted to relational data as described in [10], and the text attributes are then indexed. The DBLP data, which is in XML format, has 881,867 tuples containing information on publications, authors, titles, and pulishers. Selected text attributes (see [10] for details) are extracted and indexed. The FoodMart data, an OLAP database, stores information on products, customers, etc., and contains 428, 049 tuples. In the sequel, we call each text attribute (or text node) indexed a term.

Our implementation was done purely in Java. Apache Lucene[5], an open-source full-text search engine, was used to index the data sets. For each data set, we built two indices. The first one considers each term as a unit (a *document* in Lucene's terminology) for indexing, while the second one indexes tokens, which are obtained by tokenize the terms using whitespaces as delimiters. The token index is used for performing token expansion. In our experiments, we expand the queries by finding tokens in the index with similar spellings as the query token using Lucene's approximate string matching facility. The expansion distance is defined as string edit distance. In a more general setting, one can employ any fuzzy string matching techniques [16] and/or entity matching using WordNet.

All the experiments were conducted on an IBM Linux server with a 3.0GHz Intel Dual Core processor, 4GB of RAM, and 2TB SATA HD RAID (Level 5).

In order to systematically study the behavior of the proposed algorithms, and to minimize the subjectivity in the experimental evaluation, the test queries are generated by randomly sampling from the data and varying a number of parameters as shown in Table 1. Specifically, the queries are generated as follows. For each query to be generated, we sample t terms from the data set, and for each sampled term, we keep only h contiguous tokens contained in that term. For each character in the sampled tokens, a spelling error is introduced with probability s. We then inject some irrelevant words (words that do not appear in the data set) between the tokens to test the robustness of the proposed algorithms w.r.t. "noises" caused by irrelevant words or punctuation marks in the queries. The number of words injected is an integer uniformly distributed in $[0, d]$. We believe that our query generation methods reflect to some extent the applications we target at where queries can be long and dirty. For

[3]http://www.imdb.com/interfaces

[4]http://dblp.uni-trier.de/xml/

[5]http://lucene.apache.org

t	the number of tuples in each query
h	the number of tokens taken from each sampled tuple.
s	the probability of an spelling error for each character in the query
d	the maximum number of irrelevant words injected between tokens

Table 1: Parameters used for query generation

all the experiments, we use the default settings of $\alpha = 10, \beta = 0.2$, $\gamma = 0$, and $m = 5$, unless otherwise noted. The true segmentation is set to be the sampled database tokens used to generate the keyword search queries.

8.2 Segmentation accuracy

We first present the experimental results on segmentation accuracy. For a given query, let $\hat{S}$ denote the set of segments resulting from our algorithms, and S denote the true set of segments. Accuracy is then defined as follows:

$$Accuracy = \frac{|S \cap \hat{S}|}{|S|}.$$

We start with the results for the dynamic programming algorithm (Algorithm 2) presented in Section 4.2. In the experiments, we use three different classes of queries: *short queries*, *medium-sized queries*, and *long queries*. Short queries are generated using parameter values $h = 3$ and $t = 2$, resulting in up to 6 tokens per query. Medium-sized queries are generated with $h = 3$ and $t = 5$, providing a query length of up to 15. Parameter settings of $h = 3$ and $t = 10$ are used to generate long queries with length up to 30. For generating all the queries, s=0.01, and d=0. It is worth pointing out here that since we focus on the problem of query cleaning in real-life applications, the queries we are dealing with here are up to a magnitude longer than the typical settings described in previous works. All the experimental results presented here are based on the average of 100 queries.

Figure 2 illustrates the segmentation accuracy for the three data sets we experimented with. All three data sets have comparable accuracy for all three classes of queries, although the performance on IMDB and FoodMart is slightly better than that on DBLP. Nonetheless, in all cases, the accuracy is above 80%. In the sequel, we will mainly show the experimental results on the DBLP data.

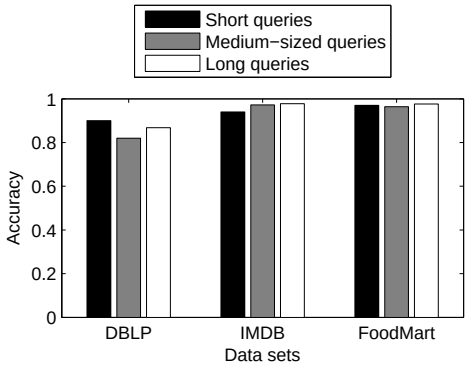


Figure 2: The segmentation accuracy for different data sets

Figure 3 shows the effect of data size on the segmentation accu-

racy. The data sets of different sizes are obtained by sampling the DBLP data set with various sample rates ranging from 0.1 (10% sample) to 1 (the whole data set). Those generated data sets are then independently indexed and queried. As can be seen from the graph, the segmentation accuracy for all three classes of queries decreases with increasing data size. Longer queries tend to result in more segmentation errors mainly because there are more segment boundaries involved. Note that, for the whole range of data size settings, the accuracy level stays above 85% for all types of queries. As will be discussed in the sequel, with top-k segmentation, we are able to achieve even higher accuracy.

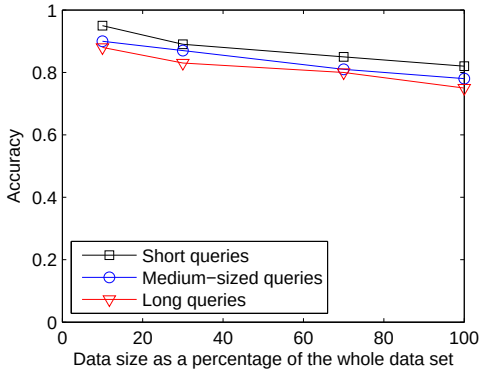


Figure 3: The effect of data size on the segmentation accuracy

The effect of spelling errors on the segmentation accuracy is shown in Figure 4. As expected, the accuracy deteriorates with increasing probabilities of spelling errors s. Note that the worst case scenario, $s = 0.1$ is very unlikely to happen in practice, as this setting actually says that for every character in *every* token, there is a 10% probability that the spelling is wrong. This is different from the case where some words are spelled entirely wrong, and the others are all spelled correctly. For a long query, our setting translates to many wrongly spelled tokens, which will certainly bring an increase in errors.

We now study the effect of d (query distance) on the segmentation accuracy. We fix $h = 3$ and $t = 2$, and vary d. The results are shown below.

Query Distance	0	1	2	3
Accuracy	0.88	0.86	0.85	0.65

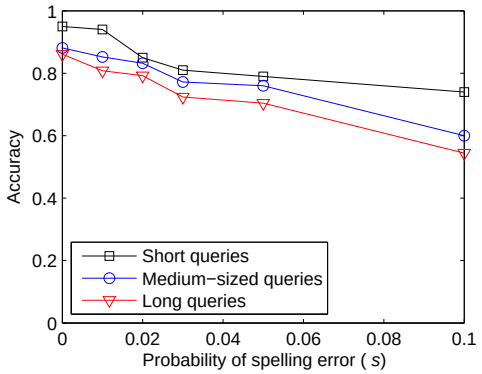


Figure 4: The effect of spelling errors in queries on the segmentation accuracy

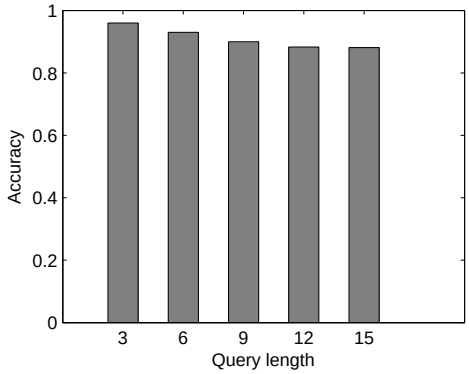


Figure 5: The effect of query length on the segmentation accuracy

As expected, the accuracy decreases when d increases. This is because the more irrelevant words are added between supposedly adjacent tokens (based on the underlying data), the more difficult it is to correctly put them into one segment.

We show the effect of query length on the accuracy in Figure 5. We fix h at 3, and vary t, the number of terms per query, from 1 to 5. The query length then varies between 3 and 15. As discussed earlier, longer queries have more boundaries to segment, and are thus more prone to errors.

In order to assess the impact of different choices of the model parameters used in the scoring function, we first vary α (sensitivity to query distance) and β (sensitivity to spelling errors/semantic differences) with γ (preference to long segments) fixed at 0. The accuracy results are shown below. We then vary γ from 0.5 to 1, with $\alpha = 10$ and $\beta = 0.2$. The accuracy remains 0.94 for all values of γ. In the experiments, $h = 2$, $t = 3$, and $s = 0.02$.

	$\alpha = 4$	$\alpha = 8$	$\alpha = 10$	$\alpha = 14$	$\alpha = 16$
$\beta = 0.1$	0.92	0.94	0.94	0.94	0.91
$\beta = 0.15$	0.92	0.94	0.94	0.94	0.91
$\beta = 0.2$	0.92	0.94	0.94	0.94	0.91
$\beta = 0.25$	0.89	0.91	0.91	0.91	0.88
$\beta = 0.3$	0.88	0.90	0.90	0.90	0.87

It is evident that our algorithm achieves consistently good performance, and is very robust with respect the choice of the above model parameters.

In order to study the effect of the expansion factor m used to expand a query token to tokens in the database, we vary its value from 1 to 6, and record the segmentation accuracy. The table below shows the results on the short queries with spelling error probability $s = 0.02$. Similar trends are observed for medium-sized and long queries. Notice in the table that the accuracy improves when we increase m, with the rate of improvement slowing down when m gets larger. The reason is that by increasing the expansion factor, we allow more similar database tokens to be included as candidates, so that the possibility of coming across the "right" token gets higher. However, the marginal benefit diminishes as the expansion factor increases, and the accuracy tends to be stable once m gets past a certain value. Note that the computational cost of segmentation increases with the expansion factor; therefore, large values of the factor should be avoided if they do not bring significant benefits to the accuracy. For our settings, an expansion factor of 5 is ideal.

Exp. fac.	1	2	3	4	5	6	7
Accuracy	0.65	0.77	0.86	0.91	0.94	0.94	0.94

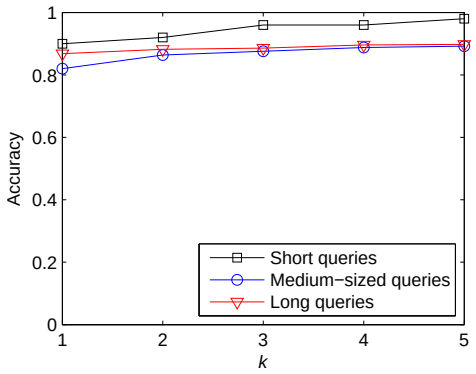


Figure 6: The effect of K on the segmentation accuracy in top-k query segmentation

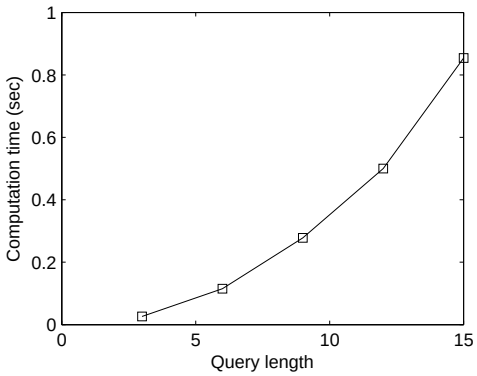


Figure 8: The effect of query length on the computation time for segmentation

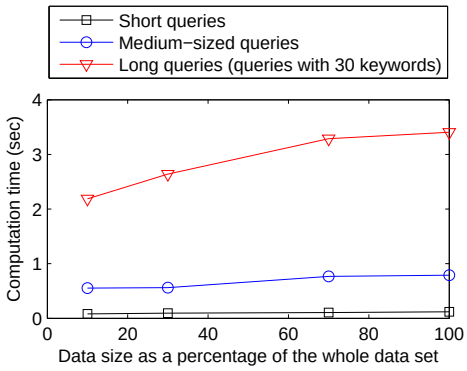


Figure 7: The effect of data size on the computation time for segmentation

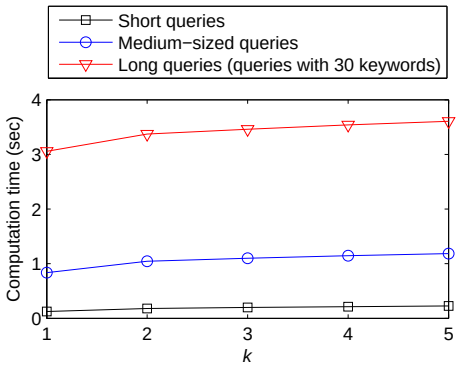


Figure 9: The effect of k on the computation time in top-k query segmentation

We now evaluate the impact of top-k segmentation on the accuracy. As shown in Figure 6, for all classes of queries, there is a healthy increase in accuracy as we increase k. For short queries, when $k = 5$, the accuracy is as high as 98%. Even for long queries, the accuracy approaches 90% as we increase k, demonstrating the high effectiveness of the top-k segmentation algorithm.

8.3 Efficiency

We measured the time required for computing the segmentation with the dynamic programming algorithm (Algorithm 2) for different data sizes, and the results are presented in Figure 7. The data sets of different sizes are obtained based on the DBLP data set in the same way as explained earlier in this section. As can be seen from the graph, the computation time required increases slowly (sub-linear) w.r.t. increasing data size. The "long queries" may require several seconds to clean, but note that they correspond to queries typically consisting of 30 keywords. We expect such cases to be rare in interactive applications. If those long queries are indeed intended to be interactive, we can rely on streamed query processing as shown in Figure 10(a), which takes less than 0.2 second per token. The "short queries", which enjoy sub-second performance, have typically 6 keywords and should be more common in real applications.

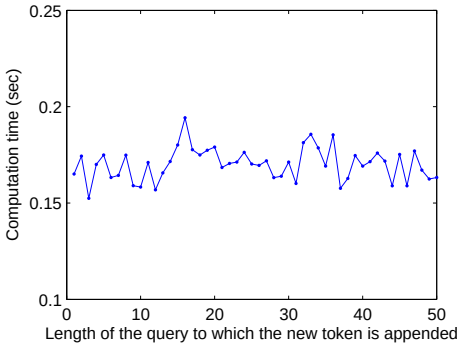
Figure 8 shows the effect of query length on the computation time. The analysis in Section 4 indicates that computation time is polynomial w.r.t. the length of the query. Empirically, as shown in Figure 8, the order of this polynomial is low, making our proposed algorithm very scalable.

For top-k segmentation, we show the impact of the choice of k on the computation time in Figure 9. We vary k from 1 to 5. As evident from the graph, the algorithm scales well w.r.t. k for all classes of queries.

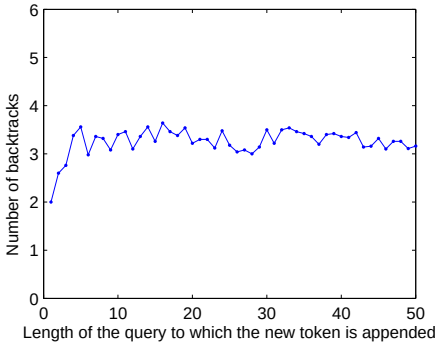
To study the performance of the incremental segmentation algorithm (Algorithm 3), we add tokens one by one to queries in an incremental fashion, and record the computation time as well as backtracks required in the algorithm for each appended token. The results are shown in Figure 10. Although theoretically, in the worst case scenario, the computation time and the number of backtracks required for each token added are linear w.r.t. the length of the existing query, in practice, as evidenced by Figure 10, the time and the number of backtracks required are almost constant (less than 200 milliseconds response time) irrespective of the query length. This supports Theorem 4. Also note that the memory usage is linearly proportional to the number of backtracks done during the incremental segmentation. By Figure 10, since the number of backtracks are bounded, the memory consumption is also bounded. Thus the incremental algorithm makes user-interacive query cleaning possible.

8.4 Search space reduction

Existing keyword search algorithms usually suffer from the search space explosion problem. A salient feature of our proposed algorithms is that significant search space reduction can be achieved,



(a) Computation time



(b) Number of backtracks

Figure 10: The effect of existing query length on the performance of incremental segmentation

which can greatly benefit the remaining phases in keyword search and tuple join network generation. The search space for simple keyword search using Lucene's built-in query capability is defined as the number of terms returned when the whole query is evaluated against the indexes. In this case, OR semantics are used by Lucene to perform the search. Since our algorithms can segment the query, separate keyword queries can be performed. We define the search space as the sum of the number of terms returned for each segment of the query, and we calculate the ratio between this reduced space and the original space. The results are shown below.

	DBLP	IMDB	FoodMart
Short	0.0003	0.0343	0.1349
Medium	0.0101	0.0007	0.1293
Long	0.0103	0.0023	0.0682

It is evident that significant search space reduction can be achieved. For example, in the case of short queries on the DBLP data, the reduction ratio is 0.0003, meaning that the reduced space is only 0.03% of the original space. For smaller data sets such as FoodMart, the reduction is not as significant, but the reduction ratio is still close to 10%.

9. CONCLUSIONS AND FUTURE WORK

We have proposed the problem and solutions of query cleaning for database keyword search queries. The cleaned query is more relevant to the database (through spelling correction and semantic translation), and concise in length (by means of segmentation). A scoring function is introduced to quantify the quality of the cleaned query in terms of the modifications made and its IR value with respect to the database content. We presented the optimal query cleaning algorithm using dynamic programming. We further extended the query cleaning algorithm to perform optimal incremental cleaning of streamed keywords, and optimal top-k query cleaning. All algorithms have been implemented and thoroughly evaluated against real-life data sets including the IMDB and DBLP data sets. We have demonstrated that our algorithms offer high degree of accuracy for a variety of queries and data sets, and is capable of handling very large queries and infinite keyword streams with subsecond performances, even for very large databases. We have also demonstrated that the query cleaning phase significantly reduces the search space for subsequent search algorithms.

As future work, we would like to extend the query cleaning problem to generate multiple distinct queries from a given body of text. Another item of future work is a self-learning and personalized query cleaning algorithm which can learn from the user behavior and preference when performing spelling corrections, semantic translation and query token segmentations.

10. REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: enabling keyword search over relational databases. In *SIGMOD*, pages 627–627, 2002.

[2] Ricardo Baeza-Yates. *Modern Information Retrieval*. Addison-Wesley, 1999.

[3] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[4] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms in middleware. In *PODS*, 2001.

[5] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: ranked keyword search over xml documents. In *SIGMOD*, pages 16–27, 2003.

[6] Vagelis Hristidis, L. Gravano, and Yannis Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[7] Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, and Divesh Srivastava. Keyword proximity search in xml trees. *TKDE*, 18(4):525–539, 2006.

[8] Vagelis Hristidis and Yannis Papakonstantinou. Discover: keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[9] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[10] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. SPARK: top-k keyword query in relational databases. In *SIGMOD*, pages 115–126, 2007.

[11] Imran R. Mansuri and Sunita Sarawagi. Integrating unstructured data into relational databases. In *ICDE*, page 29, 2006.

[12] Alexander Markowetz, Yin Yang, and Dimitris Papadias. Keyword search on relational data streams. In *SIGMOD*, pages 605–616, 2007.

[13] George A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[14] Sunita Sarawagi and William W. Cohen. Semi-markov conditional random fields for information extraction. In *NIPS*, 2004.

[15] W. J. Teahan, Rodger McNab, Yingying Wen, and Ian H. Witten. A compression-based algorithm for chinese word segmentation. *Comput. Linguist.*, 26(3):375–393, 2000.

[16] Esko Ukkonen. Approximate string-matching over suffix trees. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, pages 228–242, 1993.

[17] Ping Wu, Yannis Sismanis, and Berthold Reinwald. Towards keyword-driven analytical processing. In *SIGMOD*, pages 617–628, 2007.