Information Discovery across Multiple Streams

Vagelis Hristidis Oscar Valdivia Michail Vlachos Philip S. Yu

School of Computing and Information Sciences
Florida International University
{vagelis, oscar.valdivia}@cis.fiu.edu

IBM T. J. Watson
Research Center
vlachos@us.ibm.com

Dept. of Comp. Science
University of Illinois at
Chicago
psyu@cs.uic.edu

ABSTRACT

In this paper we address the issue of continuous keyword queries on multiple textual streams and explore techniques for extracting useful information from them. The paper represents, to our best knowledge, the first approach that performs keyword search on a multiplicity of textual streams. The scenario that we consider is quite intuitive; let's assume that a research or financial analyst is searching for information on a topic, continuously polling data from multiple (and possibly heterogeneous) text streams, such as RSS feeds, blogs, etc. The topic of interest can be described with the aid of several keywords. Current filtering approaches would just identify single text streams containing some of the keywords. However, it would be more flexible and powerful to search across multiple streams, which may collectively answer the analyst's question. We present such model that takes in consideration the continuous flow of text in streams and uses efficient pipelined algorithms such that results are output as soon as they are available. The proposed model is evaluated analytically and experimentally, where the ENRON dataset and a variety of blog datasets are used for our experiments.

Keywords: streams, keyword search, correlation, continuous queries, real-time search

1. INTRODUCTION

Nowadays data is omni-present as virtually everyone has access to the Internet; however knowledge is still very sparse. While great amounts of information are constantly provided by multiple sources, individuals and corporations are now facing a new challenge; that of ‘distilling’ the useful information.

This work explores techniques for extracting useful information from a collection of text streams. The paper represents, to our best knowledge, the first approach that performs keyword search on a multiplicity of textual streams. The scenario that we consider is quite intuitive; let’s assume that a research or financial analyst is searching for information on a topic, continuously polling data from multiple (and possibly heterogeneous) text streams, such as RSS feeds, blogs, etc. The topic of interest can be described with the aid of several keywords. Current filtering approaches would just identify single text streams containing some of the keywords. However, it would be more flexible and powerful to search across multiple streams, which may *collectively* answer the analyst’s question. Clearly, in order to collect meaningful results, the textual streams that may contain the desired answer need also be correlated (e.g., refer to same class of events, as specified by the query). The advantage of the above approach is that portions of the posed query may appear on different streams, which are aggregated to form the desired answer set.

Our methodology shares apparent commonalities with two other areas. The first one is keyword search on databases and the second is subscription/alert services (e.g., Google Alerts). We briefly elaborate on the differences with respect to these areas.

Keyword search on databases provides support for discovery of associations between the query keywords in a structured or semi-structured database; the keywords of the query do not have to be present in the same document, but can reside at different documents. However, the data sources are inherently static or updated in a batch fashion, and there is no notion of streaming and evolving textual data. The posed queries address only a specific time snapshot of the database. So, keyword search techniques are not designed to efficiently consider the incremental data additions and removals of streaming data, or even to progressively update correlations between the modified data sources.

Alert systems over text sources, on the other hand, can provide support for streaming sources. The result in this case is any instance of a stream that contains all query keywords within a specified time span. However, each stream is typically processed separately and the execution is equivalent to independently posing the continuous query on each of the streams. Notice that inter-correlations between different sources are ignored, and all keywords of the query are expected to reside on the same stream. However, considering the associations between the textual sources is very important, not only for limiting the false hits of a keyword search algorithm (this will be explained more below), but also for allowing extended search capabilities, where fragments of the posed query can exist within different text streams.

Our work bridges the above two technologies, by facilitating keyword search over a time span on multiple textual streams, taking their correlations into account. In particular, we solve the problem of answering a keyword query on a collection of text streams, where a result is defined as a combination of events from a set of correlated streams such that these events collectively contain all the query keywords. Our methodology is illustrated in Figure 1. The proposed system allows the inclusion of the inherent temporal dimension into the problem, enabling the execution of more complex keyword queries with temporal constraints, where the keywords don't have to reside in the same stream, but can be distributed over different streams.

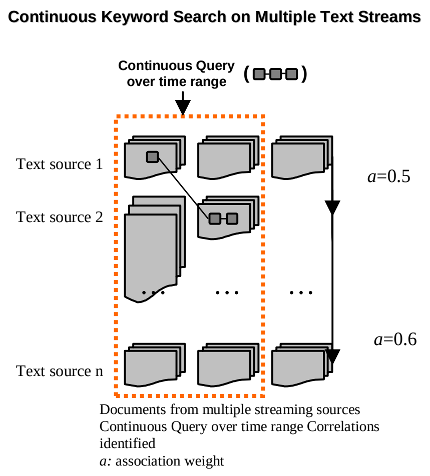

Continuous Keyword Search on Multiple Text Streams
Continuous Query over time range
Text source 1
Text source 2
Text source n
a=0.5
a=0.6
Documents from multiple streaming sources
Continuous Query over time range Correlations identified
a: association weight

Figure 1: Overview of the proposed methodology.

In order to avoid multiple spurious matching between streams of data that are unrelated to each other, we also impose the additional constraint for the results to exist within "sufficiently" correlated streams of data. For instance, given the query {arbitrating, contract, problem} and two text streams that monitor email threads, if email stream *A* mentions the words 'arbitrating' and 'problem' and emails thread *B* mentions 'contract', but stream B is unrelated to *A* (different participants and topic), then the combination of streams *A* and *B* is not a good answer to the query. In the course of the paper we explain how to define the stream association within a time window, and how to incrementally update their score as new text data are added or removed.

Applications of the above framework are quite extensive:

1. Subscription services, alert and recommendation systems that inform users of streams containing individual preferences.
2. Intelligent monitoring of a server's text content, either for the purposes of enforcing filtering mechanisms, or for categorizing and classifying the textual content more accurately.
3. More efficient collection, filtering and understanding of the diverse news feeds (closed captioning, CNN headlines, etc) for media people (e.g. journalists).
4. Better understanding and analysis of social or streaming networks, through correlation analysis of the textual chains of messages (e.g. chat messages, email logs, blogs).
5. Crime prevention through more efficient monitoring of unencrypted (or lightly encrypted) Internet text channels by law enforcement or government entities.

In our work, we present algorithms that can efficiently query a multiplicity of text streams given a set of keywords. Matches are identified within a specified time window and are presented in real-time, by assembling together relevant pieces of information from multiple associated streams. The high-performance of the algorithms is due to their incremental nature. Given specified user preferences, the algorithms perform a minimal amount of computation for each new stream event (new piece of data transmitted through a stream) in order to examine for a potential new result. In our solution, we segregate the problem of maintaining the association degrees between the text streams, from the problem of assembling and producing the results in a pipelined manner. Additionally, we identify and compare alternative strategies which are preferable for different problem settings. The contributions of this paper are the following:

1. We formally define the problem of keyword search across multiple text streams. We also identify the key user-defined parameters to calibrate the results.
2. We present efficient algorithms that can assemble, process and output query results in real-time, i.e., as they become available.
3. We study alternative techniques to measure associations between text streams. Efficient algorithms to maintain these associations are presented and evaluated.
4. To evaluate the quality of the results, as well as the performance of the proposed algorithms, we conduct a case study using the publicly available ENRON email [29] and Splog Blog [35] datasets, as our experimental testbed. We adapt the email dataset in our system prototype, by viewing email threads as continuous text streams. Comprehensive experimentation on the said datasets demonstrates the feasibility of our approach.

The paper is organized as follows: In Section 2, related work is presented. Section 3 introduces formal notation and description of the problem. Section 4 continues with the challenges and an overview of our approach. Our algorithms are presented in Sections 5 and 6. Experiments are covered in Section 7. We conclude in Section 8.

2. RELATED WORK

There has been a great corpus of work [2, 7, 17, 20, 21, 28] on keyword proximity search on static databases. These works follow various techniques to overcome the NP-completeness of the Group Steiner problem, to which the keyword proximity search problems can be reduced.

Goldman et al. [17] use pre-computation to minimize the runtime cost. BANKS [7] views the database as a graph and proposes algorithms to approximate the Group Steiner Tree problem. [2, 20, 21] perform keyword search on relational databases and exploit the schema properties to achieve efficient execution. ObjectRank [6], which returns single objects and not associations between objects as the above work, ranks the results of keyword queries using the authority flow factor. However, all this work assumes that the data is static and the time dimension of the problem is not taken into account. Markowetz, et al. [31], extend the work of [21] to streaming relational data, and present pipelined relational execution plans. In contrast, our work is on text streams.

Top-k ranked query works [5, 14, 38] compute efficiently the top results given ranked lists of attribute values for the set of objects, whereas top-k ranked join queries Ilyas et al. [24] compute the top results of a join given a ranking function. Different to our problem, the set of objects is static and there is no notion of time. Subscription systems [37, 13, 12] answer continuous user queries by examining documents in separation, in contrast with our work where we combine information from various streams to construct a result.

The area of query processing in streams has received much attention [1, 4, 3, 8, 9, 10, 39]. These works consider traditional queries with emphasis on the efficient aggregation, load balancing and stream dissemination. In contrast, our work focuses primarily on the stream analytic part, providing estimations on query-specific associations between the streams. Another related field of research deals with the join of streaming data [11, 16, 18, 27], which involves only exact matches between multiple streams within a time window. Such applications do not contain the notion of a ‘query’. In our setting a query can be comprised of a set of keywords, each one appearing on a different stream. Methods for discovering correlated bursty patterns and their bursty periods across text streams are found in [36]. Other work related with textual streams can be found in [15,26], which primarily address the modeling of a single textual stream, to support applications such as burst detection. There has been work on other streams topics such as data mining streams, to detect complex predicate patterns and find frequent item-sets adaptively over the stream [19, 23]; and in effectively processing continuous XPath queries over XML streams [30, 32]. But again our work is directed to the specific area of text streams.

Therefore to the best of our knowledge, none of the previous related research addresses the streaming keyword search problem. Although some of the approaches (like database keyword search) could be adapted to provide support for a streaming scenario, the expected performance is bound to be very far from real-time. We have presented a preliminary problem description in a previous poster [22].

Below, we will start the dissection of the problem; first we introduce the necessary definitions and notation that will be used throughout the paper. Table 1 summarizes this notation.

S	*Text Stream*
$\mathbf{S}$	*Set of streams*
S.*description*	*Stream S description*
S.participants	*Stream S participants*
S.events	*Stream S events*
e	*Event*
e.*t*	*Event e timestamp*
e.content	*Event e content*
$a(S_x,S_y)$	*Association weight of streams* S_x *and* S_y
r_p	*Participants correlation*
r_c	*Content correlation*
ξ_p	*Constant for significance of* r_p
ξ_c	*Constant for significance of* r_c
$G(\mathbf{S},E)$	*Undirected text stream graph of* $\mathbf{S}$
N_{kw}	*Number of non-stop keywords in an association window*
$\boldsymbol{x}$	*Vector of all keyword(*N_{kw}*) presence (0 for absence, 1 for presence) for S*
μ_x	*Average value of* S_x
σ_x	*Standard deviation for* S_x
p_x	*Pressure of* S_x
q	*Continuous keyword query*
q.keywords,	*Query q keywords*
q.matchingWindow	*Query q user define matching window length*
q.associationThreshold	*Query q user define association threshold*
T	*Event tree*

Table 1: Notation Used.

3. NOTATION AND DESCRIPTION OF THE PROBLEM

Definition 1 [*Text Stream***]** A *text stream S*:=(S.*description, S.participants,S.events*) consists of a *description*, an (optional) list of participants and a continuous and asynchronous sequence of events *S.events:=(*e_1,e_2,…*)* of the form *e:=(*e.*t, e.content*), where timestamp *e.t* denotes the time occurrence of an event and content *e.content* is a text string associated with the event. Therefore, a stream $S \in A^* \times (R \times A^*)^N$, $A = \{a,b,..z\}^*$, and an instance of it is S_i:=(S_i.description, S_i.participants,(($S_i.e_{i1}.t$, $S_i.e_{i1}$.content), ($S_i.e_{i2}.t$, $S.e_{i2}$.content),…)), where the notation e_{ij} describes the event *j* of stream S_i . The subscript *i* may be omitted during the course of the paper when the stream reference is not ambiguous in the related context. □

The description *S.description* is a text string providing a concise explanation of the stream content (e.g., the description element of the channel in RSS [34]. For example for a news feed from CNN, this field can hold the value: "CNN news", while an event *e* can contain the title of a news event. The *participants'* field is empty in the case of a news stream, however can be utilized when transmitting chat data (e.g. chat session) or email data. In the case of chat data an event is a single message of a participant of the chat session. Timestamp is the time the message was submitted and content is the text of the message. (Content could also include the sender of the message, but this complicates things since the list of participants of a stream is stored separately as we explain later.) The above notation only attempts to provide a high level *abstraction* of the problem, without going into details of the actual structural data organization into XML, which is not within the scope of this work.

We assume that there exists a set $\boldsymbol{S}=\{S_1,\ldots,S_p\}$ of p text streams. So for example, p chat sessions occur concurrently or p RSS feeds are transmitted online. The *association weight* $a(S_x,S_y)$ between two text streams S_x, S_y denotes how relevant these text streams are. The definition of this relevance depends on the particular application. For example, given stream of emails or chat sessions we can weigh into the computation factors such as common participants or words between two streams, and formally define it as provided in following equation:

$$a(S_x,S_y) = \xi_p \cdot r_p(S_x,S_y) + \xi_c \cdot r_c(S_x,S_y) \qquad (1)$$

where r_p and r_c describe the degree of correlation between stream *participants* and stream *content*, respectively. ξ_c, ξ_p are constants that capture the relative significance that the user is willing to provide to either of the two correlations. The constant ξ_c can be equal to zero, in the case where no participants are involved. Notice that higher association weight $a(S_x,S_y)$ denotes stronger relevance between S_x, S_y. While for the remainder of the paper we focus on the specific definition of association, this does not limit the generality of our approach, since alternate definitions of correlation [25] or similarity, could also be adapted without significant modifications.

Given a set $\boldsymbol{S}$ of text streams and the association weights between them, we construct the undirected *text stream graph* $G(\boldsymbol{S},E)$ by creating a node for each text stream in $\boldsymbol{S}$ and an edge with weight $a(S_x,S_y)$ between each pair S_x, S_y of text stream nodes with nonzero association weight $a(S_x,S_y)$.

One way to compute the content or the participants' correlation between two streams is using the Jaccard coefficient. For instance, the participants' correlation is:

$$r_p(S_x,S_y) = \frac{S_x.participants \cap S_y.participants}{S_x.participants \cup S_y.participants}.$$

An alternative way to compute the content correlation, which we used in our experiments, is viewing each text stream as a set of words (using a thesaurus to handle stemming and synonymy). Then the content correlation $r_c(S_x,S_y)$ measures the degree of association between the sets of words in S_x and S_y. We measure the degree of content similarity between two streams S_x, S_y using Pearson's correlation. The derivation of the correlation, however, can be simplified given the binary representation of the feature data. We provide the new simplified formula subsequently.

Words	S_{94332}	S_{78992}	S_{13920}	**Words**	S_{94332}	S_{78992}	S_{13920}
Dabhol	1	1	0	*Firm*	0	0	1
Arbitrating	0	1	0	*Obligations*	0	0	1
Details	1	0	0	*Krishna*	1	0	0
Give	1	0	0	*Extraordinary*	0	1	0
Terminating	1	0	0	*Unlikely*	0	1	0
India	0	1	0	*PPA*	1	0	0
Contract	0	1	1	*Lose*	0	1	0
Performance	0	0	1	*Protection*	1	1	0
…	…	…	…	…	…	…	…

Table 2: Words to streams matching matrix.

Let's assume that we have encountered N_{kw} words (not including stop-words such as "in", "and", etc) within the *association window* (user specified window of interest, for the purpose of computing the association weights) with temporal length |*associationWindow*|. Using tabular notation we can express the textual content of each stream by assigning one row per word and one column per stream (as shown in Table 2 for Example 1). Presence of a word on a stream is indicated by one and absence by zero. Therefore, the content of stream S_x is transcribed into a vector (a column in Table 2) $\mathbf{x} = [x_1 x_2 \ldots x_{Nkw}]^T$, $x_i \in \{0,1\}$. Now we can express the content correlation r_c between two streams S_x, S_y as:

$$r_c(S_x,S_y) = \frac{\sum_i (x_i-\mu_\chi)(y_i-\mu_y)}{N_{kw}\sigma_x\sigma_y} = \ldots = \frac{\sum_i x_i y_i / N_{kw} - \mu_x\mu_y}{\sigma_x\sigma_y}$$

which leads to the right-hand expression after elementary calculations. μ_x and σ_x are the average value and the standard deviation for S_x respectively. Since we have transformed our streams into binary content we can derive a simpler version of the stream correlation. The variance of a binary vector can be expressed in a simpler form using only the sum of the binary vector:

$$\begin{aligned}\sigma_x{}^2 &= \frac{\sum_i (x_i-\mu_x)^2}{N_{kw}} = \frac{\sum_i x_i{}^2}{N_{kw}} - 2\frac{\sum_i x_i\mu_x}{N_{kw}} + \frac{\sum_i \mu_x{}^2}{N_{kw}} = \frac{\sum_i x_i{}^2}{N_{kw}} - 2\mu_X^2 + \mu_X^2 \\ &= \frac{\sum_i x_i{}^2}{N_{kw}} - \mu_X^2 = \frac{\sum_i x_i}{N_{kw}} - \mu_X^2 = \frac{\sum_i x_i}{N_{kw}} - \left(\frac{\sum_i x_i}{N_{kw}}\right)^2 = \frac{\sum_i x_i}{N_{kw}}\left(1-\frac{\sum_i x_i}{N_{kw}}\right)\end{aligned}$$

We denote:

$$p_x = \sum_i x_i / N_{kw},\ \ p_y = \sum_i y_i / N_{kw} \ \text{ and } p_{xy} = \sum_i x_i y_i / N_{kw}$$

We call the notation p_x the stream *pressure* of S_x, as it indicates how strong is the presence of stream S_x in the universe of words N_{kw} within the examined window. Using the above notation the expression of the content correlation is simplified to:

$$r_c(S_x,S_y) = \frac{p_{xy} - p_x p_y}{\sqrt{p_x(1-p_x)p_y(1-p_y)}} \qquad (2)$$

We observe that essentially the content correlation depends on three parameters: (i) the total number of discrete words within the examined window, (ii) the number of discrete words of each stream ($p_x \times N_{kw}$ and $p_y \times N_{kw}$) and (iii) the number of common words ($p_{xy} \times N_{kw}$) between streams.

Example 1: Figure 2 depicts a real timeline snapshot of the ENRON email database (used in our experiments), which captures a small subset of email events. As explained in Section 7, a stream corresponds to an email session (sequence of forward and reply emails). For the streams of Figure 2 we can create the matrix notation of Table 2 that can assist in computing the stream correlations.

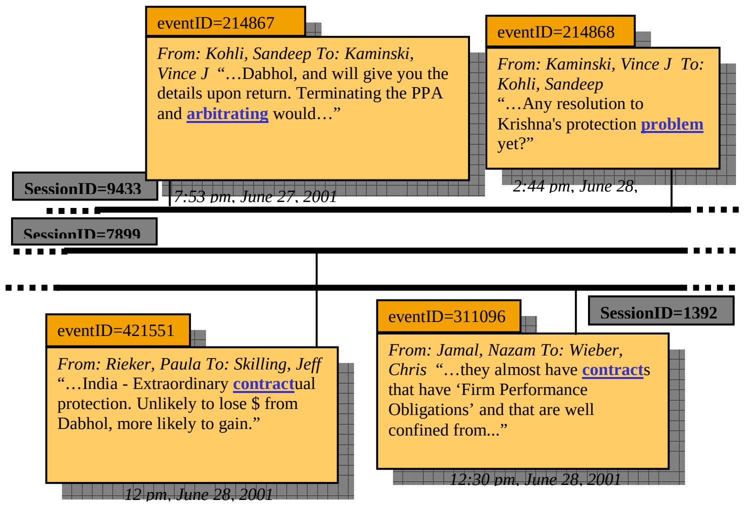

eventID=214867
From: Kohli, Sandeep To: Kaminski, Vince J "…Dabhol, and will give you the details upon return. Terminating the PPA and arbitrating would…"
7:53 pm, June 27, 2001
eventID=214868
From: Kaminski, Vince J To: Kohli, Sandeep "…Any resolution to Krishna's protection problem yet?"
2:44 pm, June 28,
SessionID=9433
SessionID=7899
eventID=421551
From: Rieker, Paula To: Skilling, Jeff "…India - Extraordinary contractual protection. Unlikely to lose $ from Dabhol, more likely to gain."
12 pm, June 28, 2001
eventID=311096
From: Jamal, Nazam To: Wieber, Chris "…they almost have contracts that have 'Firm Performance Obligations' and that are well confined from..."
12:30 pm, June 28, 2001
SessionID=1392

Figure 2: Snapshot from the ENRON emails stream.

Table 2 shows a matching matrix between keywords and streams. Using tabular notation the textual content of each stream is expressed by assigning one row per word and one column per stream. Presence of a word on a stream is indicated by one and absence by zero. Therefore, the content of stream S_x is transcribed into a vector (a column in Table 2)

The total number of words within the user specified association window is N_{kw}=50. Using Equation 2 and Table 2 we can easily compute the correlation between the two text streams [94332] and [78992]:

$$r_c(S_{94332}, S_{78992}) = \frac{\frac{2}{50} - \frac{12}{50}\frac{6}{50}}{\sqrt{\frac{2}{50}(1-\frac{12}{50})\frac{2}{50}(1-\frac{6}{50})}} = 0.34$$

Similarly we derive the remaining two correlations: $r_c(S_{94332}, S_{13920}) = -0.84, r_c(S_{78992}, S_{13920}) = 0.18$

Definition 2 [*Continuous Keyword Query***]** A *continuous keyword query* (simply called query henceforth) *q*:=(*q.keywords*, *q.matchingWindow*, *q.associationThreshold*) consists of a set of keywords *q.keywords:=(kw₁,…,kwₘ)*$\in (A^*)^N$, a user specified time window length *q.matchingWindow*, and a user specified number *q.associationThreshold*.

Threshold value *q.associationThreshold* specifies when an association between two streams becomes insignificant, that is, we ignore associations weaker than *q.associationThreshold*. A domain expert decides the values of *q.associationThreshold* and *q.matchingWindow* based on the needs and characteristics of the domain. Another alternative is for the user to provide examples of streams that are considered correlated, given which the algorithm can learn what is considered as meaningful association threshold.

The answer of a query *q* on a set ***S*** of text streams is a sequence of all event trees *T*. An event tree *T* is a tree where every node is a pair of ⟨stream $S \in \mathbf{S}$, a subset of *S.events*⟩. Intuitively, *T* is defined as a set of streams (each in a node of *T*) along with a subset of query-relevant events from each stream. If we ignore the second piece of information in the nodes of *T* (a subset of *S.events*), then *T* is a subtree of *G*. For example *T* could be $S_1(e_{13},e_{15})$-$S_4(e_{41},e_{43})$. Each event tree *T* has the following properties:

1. *T* (specifically, the events in *T*) contains all keywords in *q.keywords*, and
2. *T* is minimal, that is, we cannot remove any leaf (an event of a leaf stream or a leaf stream altogether) of *T* and still have all keywords contained, that is, there is no leaf event $e \in T$ such that $keywords(e) \subset keywords(T-e)$, and

3. the events in T occur within a window of time length *q.matchingWindow*, that is, $T.end-T.start \leq q.matchingWindow$, and
4. T is the maximum spanning tree[1] on the subgraph G_T of G that only contains the nodes/streams of T, and
5. $score(T) \geq q.associationThreshold$. □

The fourth property is used to ensure that the strongest connections between the text streams of a result are used to construct the event tree. For example, consider an event tree T consisting of the three streams. Then, T can only be S_3-S_1-S_2, but not S_1-S_3-S_2 or S_1-S_2-S_3, where the particular events for each stream are omitted for conciseness. The reason is that $a(S_1,S_2)$ and $a(S_1,S_3)$ are the two largest association weights.

Example 1 (cont'd): The strong negative correlation $r_c(S_{94332}, S_{13920})$ in our setting indicates the presence of many words in either stream, but existence of very few common words between the pair of streams. Notice the importance of the correlation computation between the streams, since it allows us to effectively filter the non-relevant answers sets.
Now, given a query *"arbitrating problem"*, the event tree $S_{94332}(e_{214867},e_{214868})$ is output, whereas for the query *"arbitrating contract problem"* the event tree $S_{94332}(e_{214867},e_{214868})$-$S_{78992}(e_{421551})$ is output. Sessions S_{94332} and S_{78992} (which conceptually correspond to different streams of events) are associated due to common words like "Dabhol" and "Protection". Clearly, the above event trees convey useful information; the first shows how arbitration problems have arisen with Krishna, and is captured by keywords existing on the same stream but on different events (emails) on the timeline. In the second event tree the various pieces of information are not only fragmented among different documents but also among different streams. Note that the event tree $S_{94332}(e_{214867},e_{214868})$-$S_{13920}(e_{311096})$ is not output, because streams S_{94332} and S_{13920} are not sufficiently associated as they share no (or few) common words. □

A salient requirement of the problem is that results should be output as they occur, which rules out any batch processing approaches, as we discuss in Section 4. Further, notice that our problem defines two different time windows: The association window (*associationWindow*) is used to define the association weights between streams, and the query matching window (*matchingWindow*) determines the maximum time span of the result-event trees. It is possible that both windows be assigned the same length depending on the application requirements. For the remaining of the paper, we will use the term "stream" instead of "text stream" for reasons of compactness.

4. CHALLENGES AND OVERVIEW OF OUR APPROACH

Given the previous work in keyword proximity search [17, 7, 2, 21], a direct approach of answering a continuous keyword query q, is to repeatedly apply a keyword proximity algorithm for each new event.
In particular, one could execute the following algorithm for every new event of the text stream set $\boldsymbol{S}$;

- First, construct a document D(S) for each text stream S that contains *S.description* concatenated with the contents of all events of S with timestamp not older than t_{now}-*q.matchingWindow*[2].
- Second, compute the text stream graph G for the association window $[t_{now}-associationWindow,t_{now}]$.
- Third, construct the document graph G_D by replacing each node S of G with $D(S)$.
- Finally, execute a keyword proximity search algorithm on G_D to compute all event trees.

The described algorithm is very expensive and inefficient because for every new event, all structures need to be initialized and recomputed from scratch. In particular, for each new event an expensive join to find all event trees has to be computed. This returns all combinations of text streams that contain all keywords and are also minimal. In addition to that, the weights of the text stream graph G are recomputed for each event. An alternative solution would be to execute this algorithm periodically (e.g., every 10 min). This approach, however, compromises the responsiveness of the system, which deviates significantly from the desired real-time.

[1] The opposite of a minimum spanning tree, since higher edge weights denote higher association in our problem.

[2] To be more formal, we should create a node for each event and connect nodes/events of the same stream with an edge with infinite weight.

In Section 5 we present an incremental algorithm that tackles the first inefficiency. In essence, we propose an algorithm that performs a join *incrementally*. This is different from previous pipelined join methods (e.g., [24]) which perform joins on a static database instance, since in our case the database instance changes over time (see Section 2 for details). We also present an analysis of the algorithm and discuss the advantages of many single queries instances instead of a single multiple query instance, as well as the trade time complexity for timeliness in the algorithm. In Section 6 we address the second issue, by presenting an algorithm that incrementally maintains the weights of the text stream graph *G*.

5. INCREMENTAL COMPUTATION OF EVENT TREES

In Definition 2, *matchingWindow* is a property of the query *q*. However, to simplify the explanation of the *tree algorithm* as well as the complexity analysis we introduce an equivalent dynamically-changing threshold *L*, which specifies time in terms of number of *query-related* events.

A query-related event is an event that contains at least one of the query keywords. That is, a query-related event e_{QR} has the following property: $q.keywords \cap e_{QR}.content \neq \emptyset$. Therefore, it characterizes events containing at least one of the query keywords. For example, if *q.matchingWindow=1 hour* and in the last hour 15 query-related events have arrived, then $L=15$. In order to simplify the analysis of the algorithm complexity we adapt the use of *L* as a measure of the window length.

The key idea of the algorithm is to maintain a forest *C* of query-related events (i.e., events that contain some query keyword), where each path from a root to a leaf represents a combination of events ordered by ascending timestamps. Each level of *C* corresponds to a single event *e* and each node of this level determines if *e* is considered in the corresponding root-to-leaf path. Each such path becomes a candidate result as we explain below.

In particular, each node on the *i*-th level of *C* [3] can either be a special node called *null* node or refer to the *(L-i)*-th latest event. For each new event *e* that contains any of the keywords *q.keywords* of *q*, we add a new level of leaves at the bottom of *C*. The candidate results (event trees) of *q* are all paths from a root to a leaf in *C*.

Intuitively, each such path in *C* represents a combination of events across a single or multiple text streams that is minimal (removing an event from the path removes a query keyword from it) and also different from all other paths. Hence, if such a path contains all keywords then it satisfies all properties of Definition 2 but the last two.
To ensure the fourth property we compute the maximum spanning tree for the graph G_T containing the text streams in the path. Then the score of *T* is computed by Equation 3.

Example 2: Consider query *q with q.keywords:= (Contract, Arbitrating), L=3, q.matchingWindow=1, and* three text streams S_1, S_2, S_3 with the following time-interleaved sequence of events (only the query-related events are shown):
e_1: in stream S_1, e_1*.content* = "The contract will be ready by…"
e_2: in stream S_2, e_2*.content* = "A very important contract"
e_3: in stream S_3, e_3*.content* = "terminating the PPA and arbitrating…"
e_4: in stream S_1, e_4*.content* = "… move the contract date…"
e_5: in stream S_2, e_5*.content* = "They prepared a contract..."

Also assume for simplicity that the association weights between S_1, S_2, S_3 are fixed to $a(S_1,S_2)=1.5$, $a(S_1,S_3)=0.5$, $a(S_2,S_3)=1.2$.

Figure 5 shows snapshots of the forest C after e_3, e_4 and e_5. Solid-line circles denote leaf nodes that correspond to a result, and dotted-line circles are the nodes that do not output a result, because the score of the event tree is less than *q.associationThreshold*. Furthermore, we cross out leaf nodes that are pruned due to the condition of line 8 of Figure 3. One can observe that only the $2^{L-1}=4$ leftmost leaf nodes are expanded, and also only *L* levels are stored at any time. Also, we note that in this example the maximum spanning trees computed in Figure 4 are simply single edges since only two stream nodes are in graph G_T for any candidate result.

[3] more formally, on the *i*-the level of a tree in *C*

```
TreeAlgorithm(input: query q, threshold L)
/*We assume that forest C of events is in steady state, that is, it has
depth L*/
1. x:=0 /*x is number of pruned nodes at each step*/
2. For each new event e do {
3.   Remove from C all roots /*that is, all instances of the oldest
               event in C*/
4.   For each of the $2^{L-1}$-pl leftmost leaves in C do
5.       Add two children: null and e          /*a pointer to e is stored*/
6.   pl:=0 /*pl is number of pruned leaves*/
7.    For each non-null leaf node u created in Line 4 do
8.    If keywords(e)⊆keywords(L-1 ancestors of u) then
      /*keywords(e)=q.keywords•e.content*/ {
9.        prune(u)
10.       pl:=pl+1
     }
11.  For each non-pruned and non-null leaf u created in line 4 do {
12.   Let p be the path starting at a root of C and
        ending at u
13.     T := getResult(p,q)
     }
  }
```

Figure 3: Tree Algorithm.

```
13a. for every $q_i$ in $q_1,…,q_n$ do
13b.     T := getResult(p,$q_i$)
```

Figure 3a

```
getResult(input: path p, query q)
1. If events in p do not contain all keywords in q.keywords then
2.     return null
3. Construct subgraph $G_T$ of G that contains all event-nodes in p
4. Compute maximum spanning tree V of text streams of $G_T$
5. Construct event tree T from V by replacing each
     text stream S by its events in p
6. If score(T)•q.associationThreshold then
7.     return T
```

Figure 4: Event tree computation algorithm.

```
2a. compute all minimal combinations of events in path
     p that contain all keywords in q.keywords
2b. for each such combination construct a new path p•
    from p by replacing the unnecessary events by null
2c. for each path p• do /*this loop ends at the end of
      getResults*/
```

Figure 4a

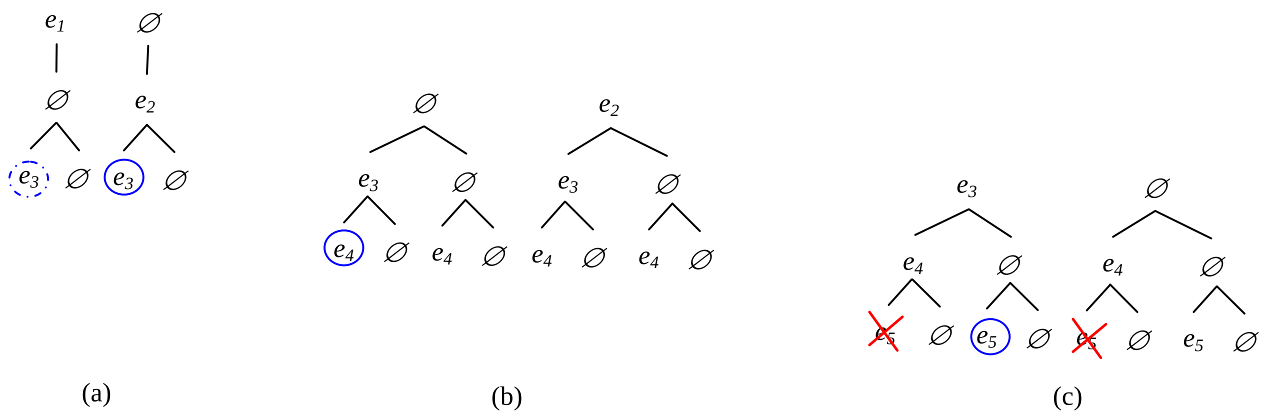


Figure 5: Snapshots of forest *C* in Tree algorithm.

The tree algorithm is described in Figures 3 and 4 where we assume the algorithm is in its steady state, that is, at least *L* query-related events have been processed. Hence, we do not show the special initializing conditions to handle the first *L* events of the streams. Consequently, *C* always has *L* levels.

Notice how in Line 4 of Figure 3 we only "expand" 2^{L-1}*-pl* leaf nodes. We subtract *pl*, which is the number of leaves pruned in the previous step, since we do not expand pruned leaves. The rationale behind 2^{L-1} is that at most 2^L leaves are needed, which is formally proven later in Theorem 1.

The pruning condition of Line 8 eliminates paths that provably cannot lead to a minimal result in the current or any future step (i.e., future event). Being more precise, if the current event does not add any keyword to the path of length *L* (that is, the current event plus the *L-1* ancestors), then no minimal result can be generated from this path.

5.1 Analysis of Tree Algorithm

Before commencing the algorithm analysis, it is important to point out that the utilized parameter *L*, takes very small values in practice (3 to 12 as we show in detail in Table 4 in Section 7) and hence the strong dependence of the algorithm on *L* does not prohibit its use in practice. For example, the event trees of Example 2 have been produced with *L=3*.

Theorem 1: *The maximum number of leaf nodes in the forest C of TreeAlgorithm that are needed to compute all query results is* 2^L.
Proof: Definition 2 requires for a results event tree *T* to satisfy *T.end-T.start≤q.matchingWindow*. *L* is a translation of *q.matchingWindow* from the time domain to the number of consecutive query-related events. Hence, equivalently, Definition 2 requires the first and last events of *T* to be contained in *L* consecutive query-related events. Given *L* consecutive events, there are 2^L combinations of subsets of them, and each such subset can potentially lead to a result. Each such subset of events corresponds to a root-to-leaf path in the forest *C*, and hence to a distinct leaf node.

Space complexity

The space complexity of the tree algorithm is $O(2^L)$, which is the space required to store a binary tree (forest in our case) with 2^L leaves (and height *L*). Notice that each node of C is just an event identifier and not a copy of the event information. Finally, we should note that due to the pruning (Lines 8-9 of Figure 4) the expected average space requirements are in practice significantly smaller (see Figure 5(c)).

Time complexity

The algorithm has three actions taking place for each query-related event.

a. Addition of new leaves (lines 4-5) with complexity $O(2^{L-1})$

b. Examination of pruning condition (lines 7-9), which also has complexity $O(L\cdot 2^{L-1})$, if we assume that given two sets of keywords containment can be decided in constant time. This is a reasonable assumption since the number of query-related keywords of an event is typically expected to be very small (1-2).

c. Validation of results (Lines 11-14) takes time $O(L\cdot 2^{L-1})$ since for each candidate result, the maximum spanning tree requires linear time (Prim's algorithm [33]) on the size of the event tree.

Therefore, the total time complexity is $O(L\cdot 2^{L-1})$ per query-relevant event. From the above discussion it is straightforward to deduce that queries containing frequent keywords (and consequently more query-related events) will require more processing per 'time-unit'.

5.2 Multiple Queries

The Tree algorithm in Figures 3 and 4 computes the results of a single keyword query. In this section we discuss the handling of multiple simultaneous continuous queries. First, we briefly present a natural extension of the Tree algorithm to handle multiple queries. Then, we explain why this approach is usually inefficient compared to executing multiple simultaneous instances of the Tree algorithm, one for each keyword query.

Suppose there are *n* continuous keyword queries $q_1,\ldots,q_n$. Let Q be a query that specifies all keywords in $q_1,\ldots,q_n$, i.e.,
$Q.keywords = \cup_i(q_i.keywords)$
and has the maximum *matchingWindow* of all queries, i.e.,
$Q.matchingWindow = max_i(q_i.matchingWindow)$.

Then, the *Multi-Query Tree algorithm* consists of the execution of the Tree algorithm for query *Q*, with the following modifications:
Replace Line 13 of Figure 3 with the lines at Figure 3a.

Also, add the following between Lines 2 and 3 of the getResult method of Figure 4 (see Figure 4a)
It can be shown that the above algorithm is complete.

Theorem 2: *The Multi-Query Tree algorithm generates all results (event-trees) for each query* q_i.

The advantage of the Multi-Query Tree algorithm is that a single forest *C* is used to answer all queries, instead of *n* separate forests $C_1,\ldots,C_n$. However, by using a 'long' query *Q* (i.e., with many keywords) instead of many "shorter" queries q_i, the depth *L* of the forest *C* is more extended. The Multi-Query Tree algorithm can only perform better than multiple instances of the Tree algorithm if there is large overlap among the keywords of the queries $q_1,\ldots,q_n$, that is, the size of *Q.keywords* is relatively small.

	Time	Space
n instances of Tree algorithm	$n\cdot L\cdot 2^{L-1}$ per query-relevant event	$n\cdot 2^{L}$
Multi-Query Tree algorithm	$l^2\cdot L\cdot 2^{l\cdot L-1}$ per query-relevant event	$2^{l\cdot L}$

Table 3: Complexities of approaches for multiple queries.

The dependence of space and time complexities on the parameter *L*, can lead to significant performance degradation for large values of *L* (unusual in practice as shown above). In particular, when *L* increases by a factor of *l* ($L' = l\cdot L$), the time and space complexities of the two approaches are shown in Table 3. Note that in the time cell of the Multi-Query Tree algorithm we multiply by *l* because the number of query-related events (the costs in Section 10.1 are per query-related event) in the Multi-Query Tree algorithm is larger by a factor of *l*.
The Multi-Query Tree algorithm can only perform better than multiple instances of the Tree algorithm if there is large overlap among the keywords of the queries $q_1,\ldots,q_n$, that is, the size of *Q.keywords* is relatively small. In that case the *l* factor described above is very small.

5.4 Trade Complexity for Timeliness in Tree Algorithm

If the real-time result creation requirement can be relaxed, the complexity can be greatly reduced. We only sketch the *grouping version* of the Tree algorithm due to space considerations. To do so, we partition the streams into groups of *b* consecutive query-related events. For example, if *b=2*, the stream *e1,e2,e3,e4* is split into the stream of groups *g1,g2*, where *g1={e1,e2}* and *g2={e3,e4}*. For each group we store the query keywords it contains along with the pairwise shortest distances (in terms of query-related events) between them. Then, a modification of the Tree algorithm is executed on the stream of groups.

The key difference is that for every candidate group tree (defined similarly to event tree) result, we have to extract the set of corresponding event trees (a group tree can create multiple event trees). Within a group, a hash map is used to efficiently map query keywords to events. The complexity of this method is $L/b \cdot 2^{L/b-1}$ and in practice can become as inexpensive as needed by choosing a large *b*. The drawback of this method is that the output of an event tree result can be delayed for up to the length of a group. Hence, the choice of *b* balances the needs of timely results and fast execution.

6. COMPUTATION OF ASSOCIATION WEIGHTS IN TEXT STREAMS

The *getResult* method of Figure 4 requires the weights $a(S_i,S_j)$ between each pair S_i, S_j of text stream nodes of the path *p*. These weights typically change over time. For example, according to Equation 1 for chat streams, for every new event all two components (participants' correlation, content correlation) may change. Note that in the description of a email events in Section 3, the participants are part of the stream description. However, they could also be part of individual events. We consider, and experimentally evaluate in Section 7, lazy and an incremental update strategies for computing the edge weights.

6.1 Lazy Strategy

One can follow a "lazy" approach and only compute the weights when needed, that is, when a minimal event tree has been constructed and we need to compute its score in order to decide if it will be output or not (Line 3 in the *getResult* method of Figure 4). Hence, in this strategy not all weights of the text stream graph *G* are maintained. The advantage of this technique is that we do not compute any "useless" weights, that is, any weights that will never be involved in a candidate result.

The disadvantages of the lazy strategy are the following: first, the 'from-scratch' computation of the pairwise weights can be expensive even if a few streams are involved in a candidate result, because all events of these streams within the association window have to be considered.

Second, the lazy execution is unaware of the events occurring that do not lead to a candidate result, and may lead to a weight computation of an edge that is guaranteed to have unchanged weight. For example, consider two consecutive candidate results r_1, r_2, both involving the edge $\langle S_i,S_j \rangle$. If the events occurring between r_1 and r_2 do not affect $a(S_i,S_j)$ (see below), then the recomputation of $a(S_i,S_j)$ was wasteful.

Clearly, the type of events that affect an edge weight $a(S_i,S_j)$ depend on the weight computation formula. Let us consider the weight formula of Equation 1 and focus on its most expensive portion, which is the content correlation component computed by Equation 2. Then, a new event $e^N \in S_i$ does not affect any edge weight, if for every keyword $e^N.kw_t$ it holds that: $e^N.kw_t \cap e_{ij}.kw_u \neq \emptyset$, for every event e_{ij} and event keyword $e_{ij}.kw_u$ with $e_{ij}.t \geq t_{now}$ - *q.associationWindow*. Therefore, the edge weights do not change if all words of event e^N already exist in the current association window.

6.2 Incremental Strategy

We now present an algorithm that incrementally maintains the complete stream graph *G* for each new event *e*. In this case, we need to consider all events (and not only the query-related, as was the case for the Tree algorithm of 5.1), since the association between two streams depends on the total set of encountered keywords and not only on the query keywords.

Considering the content correlation of Equation 2, Figure 6 presents an algorithm for its incremental maintenance. In order to facilitate incremental computation, the algorithm records the pressure p_x for every stream S_x, as well as the pressure between pairs of streams $p_{x,y}$, $S_x \neq S_y$.

```
IncrementalStreamGraphMaintain(event e of stream $S_x$)
1. Let NW be the set of new words in e with size $N_{kw} \cdot p_x^{new}$ /*words that have
       not already appeared in $S_x$ during the association window*/
2. Let EW of the set of expired words from $S_x$ with size $N_{kw} \cdot p_x^{expired}$ /*words
       That appeared in $S_x$ during the previous association window but do
       not appear during the current association window*/
3. If NW and EW are empty then return /*do nothing*/
4. $p_s = p_s^{prev} - p_x^{expired} + p_x^{new}$
5. For each stream $S_y$ <>$S_x$ do {
6.    If UpperBound($r_c(S_x,S_y)$) < q.associationThreshold
        then continue /*prune and go to next stream $S_y$*/
7.    Let NW• be the set of new words in NW for $S_y$ with size $N_{kw} \cdot p_{xy}^{new}$
        /*words in NW that also appear in $S_y$ during the current association
        window*/
8.    Let EW• be the set of words in EW also in $S_y$ with size $N_{kw} \cdot p_{xy}^{expired}$
        /*words in EW that appear in $S_y$ during the current association
        window*/
9.    $p_{xy} = p_{xy}^{prev} - p_{xy}^{expired} + p_{xy}^{new}$
10.   Compute $r_c(S_x,S_y)$ using Equation 2 /*Notice that $p_y$ does not change*/
   }
```

Figure 6: Incremental maintenance of stream graph.

We use the following notation. Suppose an event *e* of stream S_x just arrived. We refer to the association window just before (resp. after) *e* as previous (resp. current) association window. Let us denote as p_x^{prev} the pressure of stream S_x during the previous association window, $p_x^{expired}$ the number of expired keywords of S_x (i.e., keywords that appeared in the previous but not in the current association window), and p_x^{new} the number of the newly added keywords (i.e., keywords that just appeared in an event of S_x for the first time in the current association window). Similar notation is used for the number of keywords in the totality of streams (N_{kw}^{prev}, $N_{kw}^{expired}$, N_{kw}^{new}). Then, if we assume for presentation simplicity that the total number of words in the association window is constant (if it changes, the formula is modified to include the new and previous total number of words), the pressure within the current window for stream S_x is:

$$p_x = p_x^{prev} - p_x^{expired} + p_x^{new}$$

A similar expression can also be computed for the common pressure between streams S_x and S_y.

$$p_{xy} = p_{xy}^{prev} - p_{xy}^{expired} + p_{xy}^{new}$$

Assuming that there are no inherently imposed space restrictions, one can optimize performance by allocating one counter for each encountered word within the association window for each stream. This assumption is realistic when one is interested in short term stream correlations, where the association window does not have a considerable temporal duration.

We notice that we can compute an upper bound on the content correlation between streams S_x, S_y utilizing only the pressures p_x, p_y of the streams (without having to resort to computing the more expensive common pressure p_{xy}). It holds that for any binary streams $S_x=[x_1, x_2,\dots]$ and $S_y=[y_1, y_2,\dots]$:

$$\sum_i x_i y_i \le \sum_i x_i \qquad \textit{and} \qquad \sum_i x_i y_i \le \sum_i y_i$$

Therefore:

$\sum_i x_i y_i \leq \min(\sum_i x_i, \sum_i y_i)$

That is,

$p_{xy} \leq \min(p_x, p_y)$

From Equation 2, we have:

$$UpperBound\ (r_c(S_x, S_y)) = \frac{\min(p_x, p_y) - p_x p_y}{\sqrt{p_x(1-p_x)p_y(1-p_y)}}$$

When $UpperBound(r_c(S_x,S_y)) < q.associationThreshold$,[4] a candidate result can be eliminated from examination (Line 7 in Figure 6) because it definitely does not satisfy the stream association requirements set by the user. From a system management point of view, the algorithm of Figure 6 makes use of a *sliding inverted index* which stores for each keyword the list of streams that the keyword has appeared, within the current association window. The sliding inverted index can efficiently maintain the following information for each new event e of stream S_x:

- the set NW of words that have not already appeared in S during the current association window
- the set NW' of words in NW that do not appear in another stream S_y during the current association window
- the set EW of words that appeared in S_x during the previous association window but do not appear during the current association window
- the set EW' of words in EW that appear in another stream S_y during the current association window

Our current implementation of a sliding inverted index is rudimentary but fully functional and is based on the use of counters that maintain the time of occurrences of each keyword in every stream, in order to discover the currently active set of keywords. More spartan alternatives for the implementation of the inverted index are also being considered, however the experimental results of Section 7.1, are based on the basic implementation with counters.

Notice, that we assume that the association window for a stream only changes when an event from this stream occurs. This is why p_y does not change in Figure 6. We experimentally evaluate the incremental algorithm in Section 7.2.

7. EXPERIMENTS

In our experiments we used the Enron email (see [29] for a description) and Splog Blog datasets.

Enron: After cleaning the data we ended up with 217,087 distinct emails, which we partition into 147,917 email threads as follows. Two emails *u, v* belong to the same thread if all of the following apply:

- There is at least one common participant (sender or recipient) between u and v.
- The subjects of *u, v* are the same modulo “Re:”, “Fw:” prefix
- The timestamps of *u, v* differ by at most 3 months.

The emails of the dataset correspond to events in our framework, where the timestamp is the email timestamp and the content is the combination of the subject and email body. This arrangement creates a correspondence between an email thread and a text stream.

Splog: The original Splog Blog dataset [35] contains 3000 blogs. In our framework every entry on the blogs corresponds to an event, and its author and time correspond to the event’s participant and timestamp respectively. The blog description, if any, would correspond to the stream description and the blog itself would be a text stream.

We conduct two sets of experiments. The first evaluates the Tree algorithm (described in Section 5.1), while the second compares the approaches for the maintenance of the stream graph: (i) the incremental maintenance, (ii) the lazy maintenance of the edge weights. The Splog Blog Dataset contains many artificial and automatically generated

[4] If the participants’ correlation is computed first (which is simpler) then a tighter bound can be used.

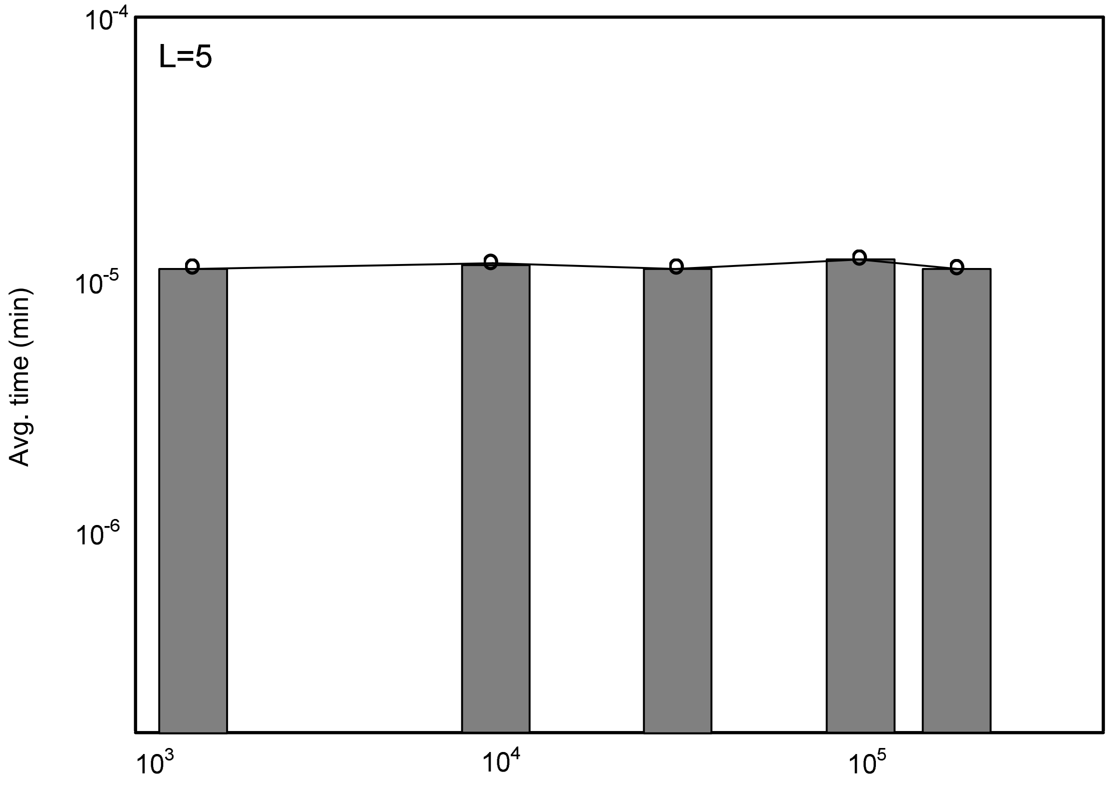
blogs, which makes the correlation computation between such blogs not useful or meaningful. Hence, we only use this dataset in the first set of experiments (Section 7.1), and assume that blogs are equally correlated.

7.1 Tree algorithm

In order to evaluate the performance of the Tree algorithm, we separate the execution of the algorithm from the edge weights calculation. (The latter is the focus of Section 7.2.) That is, the times reported in this section do not include the time for the graph maintenance, which for this experiment are already pre-computed. Additionally, since the Tree algorithm is executed for each query-related event (i.e., an event that contains at least one of the query keywords), we measure the execution time *per query-related event*. This metric can also provide a practical system calibration tool, for predicting the system performance under specific word distributions. For example, given a set of streams with a known (or predicted) average rates of query-related events, a system analysts can easily estimate whether the Tree algorithm will provide real-time responses for a given *L* (*L* is defined in Section 5.1) and number of query keywords.

Varying number of streams

Figure 7 report the average execution time in *minutes*, for different number of streams, two keywords and *L*=5. The graph suggests that the execution time remains more or less the same for different number of streams. This is expected since the time is dependant on the processing of query-events and although more streams will contribute with more query-events, their time is averaged which makes the results



similar.

Figure 7: Average execution times per query-related event for varying number of streams for Enron Dataset.

Varying *L*

Before measuring the performance for varying *L*, we discuss what are typical values of *L*. Table 4 shows the average number of query-events (emails) and time span for various values of L, two-keyword and three-keyword queries. The last column (total #results) shows the total number of result event trees for the given query and *L* across the whole stream. The results show that very small values of *L* query-related events typically cover a long time-span (hours or even days). This observation is important for maintaining the real-time profile of the algorithm.

keyword 1	keyword 2	keyword 3	L	time (hr)	#query-events	Total #results
opportunities	pipeline		6	90.66	833	7164
opportunities	pipeline		9	135.99	1249	8028
opportunities	pipeline	partnership	6	77.82	886	3584
opportunities	pipeline	partnership	9	116.73	1328	5446
settlement	pay		6	18.42	920	10080
settlement	pay		9	27.63	1380	12960
settlement	pay	India	6	19.02	889	203
settlement	pay	India	9	28.53	1334	319

Table 4: Relation of L to number of events and time span for Enron dataset.

In our first experiment we measure the query execution time per query-related event. Figure 8 shows the execution time in *msec* for four sample queries along with the keyword frequencies in the Enron and Splog Blog Datasets (i.e., number of events that contain them). Queries {EIA, estimate} and {breaker, PMT} were ran against the Enron dataset. The other queries, {summer, yoga} and {chinese, proverb} correspond to the Splog dataset.

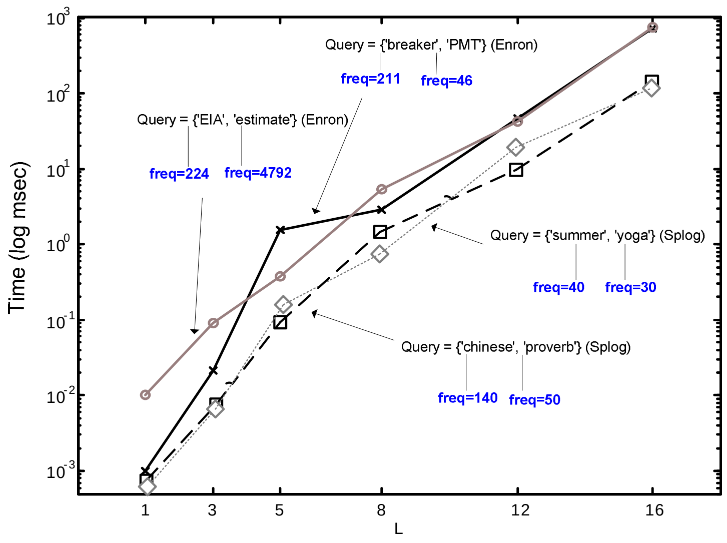


Figure 8: Execution time per query-related event for four sample queries and varying L for the two datasets.

We utilize two keywords per query and test the performance of the system on 50 continuous keyword queries. In Figures 9 and 10 we report the *average* execution time over all queries, in order to remove the bias of either very short or very large queries (containing either very infrequent or very frequent words). Clearly, the system can return results in time less than a second, even for large values of L (*L=16*), which can typically corresponds to hundreds or thousands of events and cover an extensive query time range.

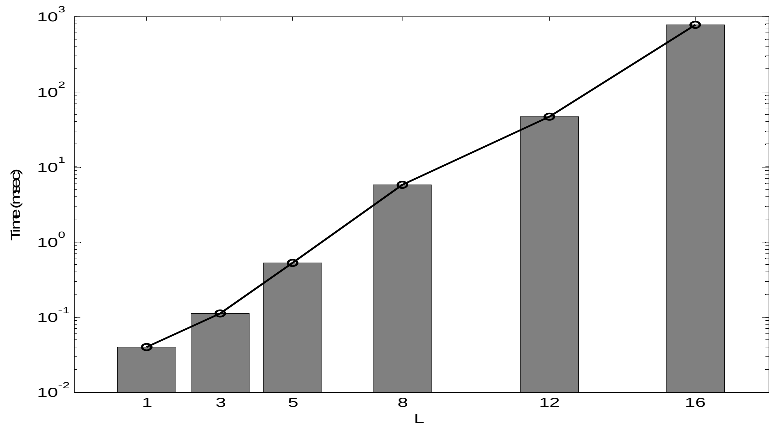


Figure 9: Average execution times per query-related event for varying *L* for Enron Dataset.

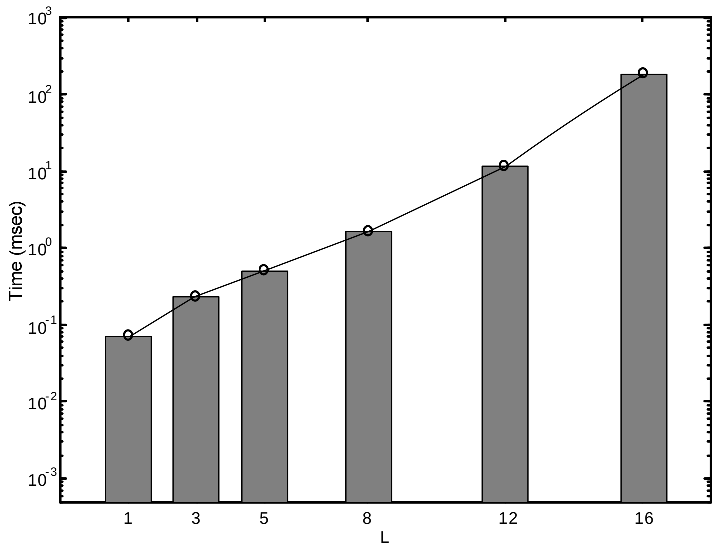


Figure 10: Average execution times per query-related event for varying *L* for Splog Blog Dataset.

Varying number of keywords

Next, we measure the effect of query keyword cardinality on the execution time. Figures 11 and 12 depicts the average execution time over 50 continuous keyword queries for three different values of L (L=5, 10, 15). These times are per query-related event, with rate naturally increasing with increasing number of keywords. The reason of prolonged execution time per query-related event is because the pruning condition of Line 8 in Figure 5 is becoming more expensive with the additional keywords. Nonetheless, the response time is still typically kept below 1 sec.

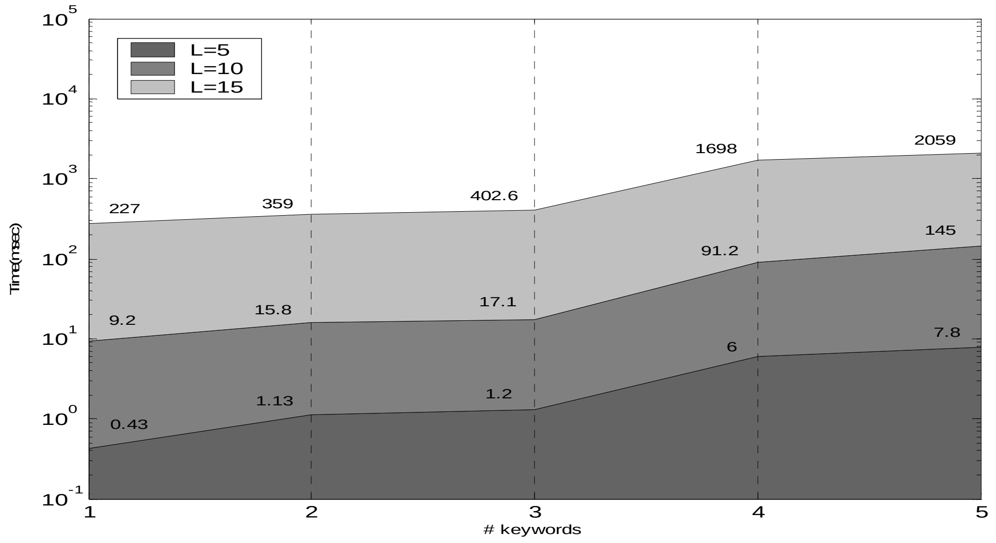


Figure 11: Average execution times per query-related event for varying number of keywords and *L* values for Enron Dataset.

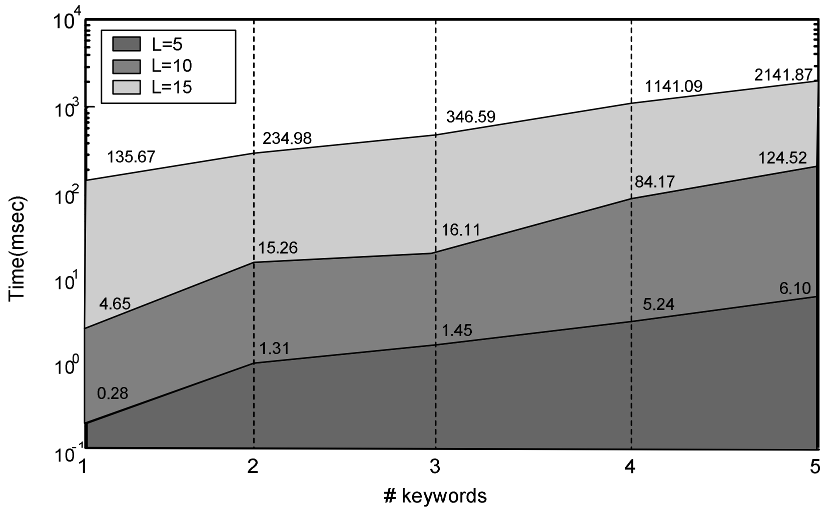


Figure 12: Average execution times per query-related event for varying number of keywords and *L* values for Splog Dataset.

Multi-Query Tree Algorithm

Figure 13 compares the Tree algorithm's average execution time of an instance of the multi-query versus two instances of a single-query. We use L=5 and varied the number of keywords per query, as well as the overlap of keywords in the multiple-query instance. As expected from our analysis in Section 5.2, the execution time for the multiple-query instance is worse than two instances of a single-query when there are no keywords in common, but improves when there are common terms in the queries.

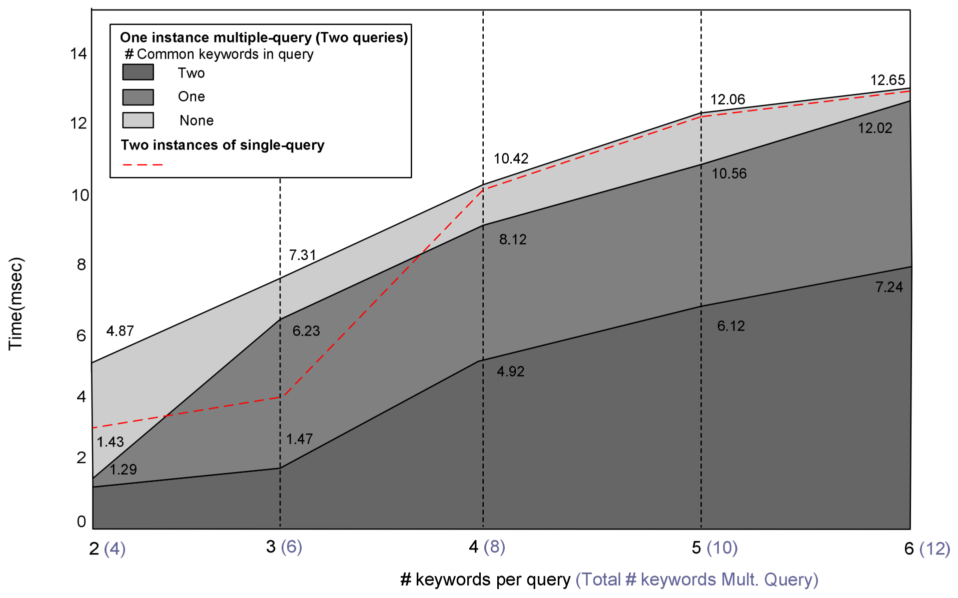


Figure 13: Average execution times per query-related event for varying number of queries and keywords for Enron Dataset

7.2.1 Stream graph maintenance

The final experiment compares the performance of the algorithms that compute the association degree between the text streams. The lazy and the incremental stream graph maintenance methods (Sections 6.1 and 6.2) compute the content correlation between the streams as defined by Equation 2. Figure 14 juxtaposes the results of these two methods for 10 queries (each being comprised of 2 keywords). The x-axis displays the average processing time for each event (here we count all events and not only query-related ones).

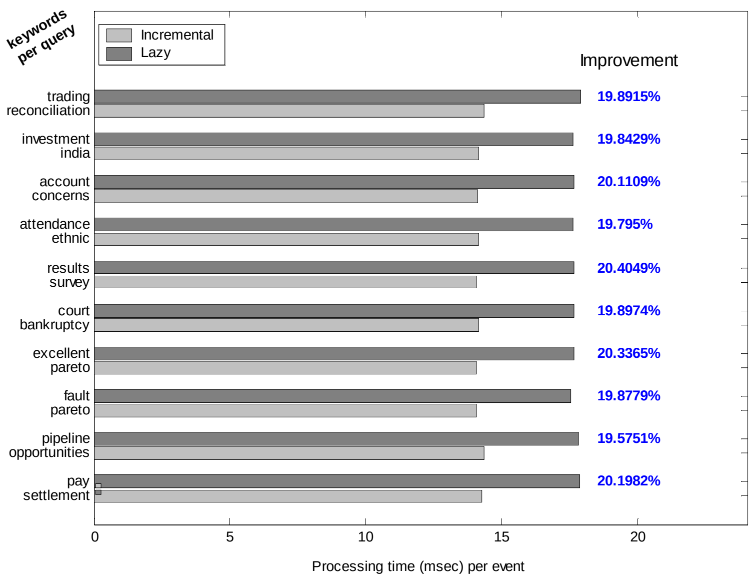


Figure 14: Comparison of stream graph maintenance techniques for Enron dataset.

We observe that the incremental technique depicts a computational advantage of about 20%. For multiple simultaneous continuous queries this improvement is even larger because the rate of query-related events increases. In particular, the incremental algorithm is executed once for each event whereas the lazy only when a candidate event tree is evaluated (in Line 3 of Figure 4). Hence, in the presence of multiple simultaneous continuous queries, the cost of lazy maintenance increases, whereas for incremental evaluation it remains constant.

8. CONCLUSIONS AND FUTURE WORK

We presented the problem of continuous keyword search on multiple text streams, which bridges the independently well-studied problems of keyword search on databases and alert services on single streams. We define a result as a tree of events from multiple associated streams, where the association is determined by the commonality of two streams, although other metrics are also possible.

We present an incremental algorithm for computing the answer set of a continuous keyword query. This algorithm performs a minimal amount of operations for each event, in essence accomplishing a streaming incremental join that utilizes partial results. We also presented and experimentally compared alternative techniques to maintain the stream graph, which stores the association weights between the streams.

As future work, we plan to investigate more complex association semantics between the streams. For example, how could one use an ontology or application specific knowledge? Additional questions of interest are how the system can automatically learn suitable values of *L*, given the approximate rate of results that we want to get for a query. For instance, in a data monitoring application, human experts may only be able to process 10 results per minute; therefore there is no need for the system to flood them with additional results that are going to be simply ignored.

9. REFERENCES

[1] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, Jennifer Widom: Characterizing Memory Requirements for Queries over Continuous Data Streams. PODS 2002

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. ICDE, 2002

[3] Ahmed Ayad, Jeffrey F. Naughton: Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. SIGMOD 2004

[4] Magdalena Balazinska, Hari Balakrishnan, Michael Stonebraker: Load Management and High Availability in the Medusa Distributed Stream Processing System. SIGMOD Conference 2004

[5] N. Bruno, L. Gravano, A. Marian. Evaluating top-k queries over Web-accessible databases. ICDE, 2002

[6] A. Balmin, V. Hristidis, Y. Papakonstantinou: Authority-Based Keyword Queries in Databases using ObjectRank. VLDB, 2004

[7] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti and S,Sudarshan: Keyword Searching and Browsing in Databases using BANKS. ICDE, 2002

[8] Mitch Chemiack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, Stanley B. Zdonik: Scalable Distributed Stream Processing. CIDR 2003

[9] Graham Cormode, Minos N. Garofalakis, S. Muthukrishnan, Rajeev Rastogi: Holistic Aggregates in a Networked World: Distributed Tracking of Approximate Quantiles. SIGMOD 2005

[10] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, Rajeev Rastogi: Sketch-Based Multi-query Processing over Data Streams. EDBT 2004

[11] A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams, SIGMOD, 2003

[12] Cristian Fiorentino, Mariano Cilia, Ludger Fiege, Alejandro P. Buchmann: Building a Configurable Publish/Subscribe Notification Service. DAIS 2005

[13] Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, Dennis Shasha: Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. SIGMOD 2001

[14] R. Fagin, A. Lotem, M. Naor. Optimal aggregation algorithms for middleware. PODS, 2001

[15] P.C.Fung, X. Yu, P. S. Yu, H. Lu Parameter Free Bursty Events Detection in Text Streams, VLDB 2005

[16] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. VLDB, 2003

[17] R. Goldman, N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina: Proximity Search in Databases. VLDB, 1998

[18] M. A. Hammad and W. G. Aref. Stream window join: Tracking moving objects in sensor-network databases, SSDBM, 2003

[19] Lilian Harada, Detection of complex temporal patterns over data streams, Information Systems, Volume 29, Issue 6, (2004) pp. 439-459

[20] V. Hristidis, L. Gravano, Y. Papakonstantinou: Efficient IR-Style Keyword Search over Relational Databases. VLDB, 2003

[21] V. Hristidis, Y. Papakonstantinou: DISCOVER: Keyword Search in Relational Databases. VLDB, 2002

[22] Vagelis Hristidis, Oscar Valdivia, Michalis Vlachos, Philip S. Yu: Continuous Keyword Search on Multiple Text Streams. Poster paper, ACM CIKM 2006

[23] Joong Hyuk Chang and Won Suk Lee, Finding recently frequent itemsets adaptively over online transactional data streams, Information Systems, Volume 31, Issue 8, (2006), pp. 849-869

[24] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid: Supporting Top-k Join Queries in Relational Databases. VLDB 2003

[25] Chris Jermaine: The Computational Complexity of High-Dimensional Correlation Search. ICDM 2001

[26] J. Kleinberg. Bursty and hierarchical Structure in Streams, SIGKDD 2002

[27] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams, ICDE, 2003

[28] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, H. Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. VLDB, 2005

[29] B. Klimt and Y. Yang. Introducing the Enron corpus. First Conference on Email and Anti-Spam (CEAS), 2004

[30] Hyun-Ho Lee, Won-Suk Lee, Selectivity-sensitive shared evaluation of multiple continuous XPath queries over XML streams. Information Sciences, Volume 179, Issue 12, (2009), pp. 1984-2001

[31] Alexander Markowetz, Yin Yang, Dimitris Papadias: Keyword search on relational data streams. SIGMOD Conference 2007: 605-616

[32] Jun-Ki Min, Myung-Jae Park, Chin-Wan Chung, XTREAM: An efficient multi-query evaluation on streaming XML data, Information Sciences, Volume 177, Issue 17, (2007), pp. 3519-3538

[33] R.C. Prim. Shortest connection networks and some generalizations. Bell Systems Technology Journal, 36:1389-1401, 1957

[34] RSS 2.0 Specification. http://blogs.law.harvard.edu/tech/rss, 2005

[35] Splog Blog Dataset. http://ebiquity.umbc.edu/resource/html/id/212/Splog-Blog-Dataset, 2007

[36] Xuanhui Wang, ChengXiang Zhai, Xiao Hu, Richard Sproat. Mining Correlated Bursty Topic Patterns from Coordinated Text Streams. KDD 2007

[37] T. W. Yan, and H. Garcia-Molina. The SIFT information dissemination system. ACM Trans. Database Syst. 24, (1999), pp. 529-565

[38] Clement T. Yu, George Philip, Weiyi Meng: Distributed Top-N Query Processing with Possibly Uncooperative Local Systems. VLDB 2003

[39] Rui Zhang, Nick Koudas, Beng Chin Ooi, Divesh Srivastava: Multiple Aggregations Over Data Streams. SIGMOD 2005

10. ACKNOWLEDGMENTS

Partly supported by NSF grants IIS-0811922 and IIS-0534530.