A System for Keyword Search on Textual 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
mvlachos@cs.ucr.edu, psyu@us.ibm.com

ABSTRACT

An increasing amount of data is produced in the form of text streams – these can be RSS news feeds, TV closed captions, emails, etc. We study the problem of answering keyword queries on multiple textual streams. We define the result of a keyword query inspired by previous work on keyword search on static databases. A result to a query is a combination of streams "sufficiently correlated" to each other that collectively contain all query keywords within a specified time span. On the algorithmic side, in this paper we focus on the component of continuously monitoring the streams and outputting results as soon as they are available.

Keywords: text streams, keyword search, continuous queries

1. INTRODUCTION

An increasing amount of data is produced in the form of text streams – these can be RSS news feeds, TV closed captions, emails, etc. We explore techniques for extracting useful information from a collection of text streams. The scenario that we consider is quite intuitive; let's assume that a professional analyst would like to continuously be informed about occurrences of a topic, a topic that can be described by several keywords. Information can be provided by various feeds in a streaming fashion. In order to motivate better our example with recent events, let us consider the case of a journalist that might be interested in tracking down people's stories in the state of Florida and how they perceive that recent hurricane events affect housing prices. Chat streams and newsgroups can provide a wealth of information on opinions of individuals. Assuming that there is a mechanism for extracting the text feeds from such sources (since they are channeled unencrypted), the interested user can set up a continuous query on a *collection* of streams of the form: `{Florida, hurricane, housing}`.

The tackled problem has common characteristics to the areas of keyword search on databases and subscription/alert services (e.g., Google Alerts). We briefly elaborate on the differences to these areas.

Keyword search on databases provides support for discovery of associations between the query keywords in a structured or semistructured database; the keywords of the query do not have to be present in the same document, but can reside at different locations. 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. That is, keyword search techniques are not designed to efficiently tackle 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 (explained below) of a keyword search algorithm, but also for allowing extended search capabilities, where fragments of the posed query can exist within different text streams.

In this work we borrow ideas from both these areas to facilitate keyword search over a time span on multiple textual streams, taking their correlations into account. 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. 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, if stream A mentions the words 'Florida' and 'housing' and stream B mentions 'hurricane', but stream B is a cocktails' recipes stream ('hurricane' is a name of a cocktail), then the combination of streams A and B is not a good answer to the query. The reason is that streams A and B are not correlated, as they discuss different domains. Due to lack of space we do not show how the stream

[*] Partly supported by NSF grant IIS-0534530.

associations within a time window are defined and computed. An overview of the architecture is provided in Figure 1.

The 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. from chat server messages or email logs).
5. Crime prevention through more efficient monitoring of unencrypted (or lightly encrypted) Internet text channels by law enforcement or government entities.

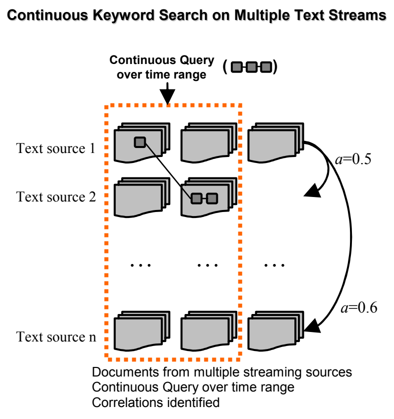
Continuous Keyword Search on Multiple Text Streams
Continuous Query over time range
Text source 1
Text source 2
Text source n
a=0.5
a=0.6
Documents from multiple streaming sources
Continuous Query over time range
Correlations identified

Figure 1: Overview of the proposed methodology.

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. 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, given specified user preferences. A high level illustration of the proposed methodology is provided in Figure 1.

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. 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 dataset, as our experimental testbed. We adapt this dataset in our system prototype, by viewing email threads as continuous text streams.

2. RELATED WORK

There has been a great corpus of work [ACD02, BNH+02, GSVG98, HGP03, HP02] 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. ObjectRank [BHP04], 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.

Top-k ranked query works [FLN01] 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. [IAE03] compute the top results of a join given a ranking function. In contrast to our problem, the set of objects is static and there is no notion of time.

Subscription systems [FJL+01,FCFP05] answer continuous user queries by examining documents in separation, in contrast to this work where we combine information from various streams to construct a result.

The area of query processing in streams has received much attention [ABB+02, AN04, CGMR05, ZKOS05]. These works tackle 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. We have presented a preliminary problem description in a recent poster [HVVY06].

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 occurrence of event and content $e.content$ is a text string associated with e. 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 current 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 [RSS05]. 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). For this situation 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 there is a set $\boldsymbol{S}=\{S_1,...,S_p\}$ of p text streams. For example, p chat sessions occur concurrently.

The *association weight* $a(S_x,S_y)$ between two text streams S_x, S_y denotes how relevant these text streams are. We do not discuss the computation of $a(S_x,S_y)$ in this paper.

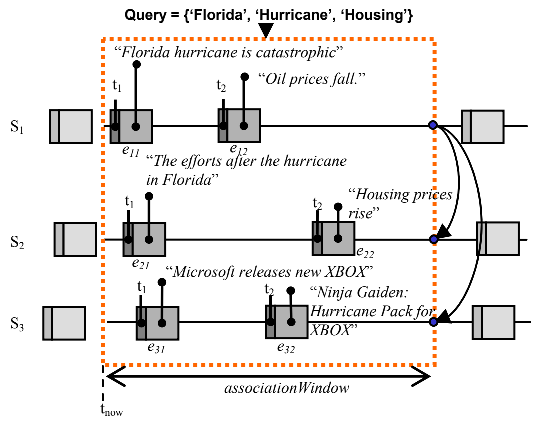
Query = {'Florida', 'Hurricane', 'Housing'}
"Florida hurricane is catastrophic"
"Oil prices fall."
"The efforts after the hurricane in Florida"
"Housing prices rise"
"Microsoft releases new XBOX"
"Ninja Gaiden: Hurricane Pack for XBOX"
S1
S2
S3
t1
t2
e11
e12
e21
e22
e31
e32
associationWindow
tnow

Figure 2: Representation of streaming textual content.

Text stream graph

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 $a(S_x,S_y)$.□

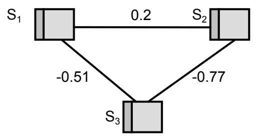

Figure 3: Text stream graph for streams of Figure 2.

Figure 3 shows a possible text stream graph for the streams of Figure 2.

Event tree

A text *stream tree* Q is a subtree of G. For example, S_1-S_2 is a text stream tree (path in this case) for the graph of Figure 3. An *event tree* T is a stream tree, where a set of events is selected from each text stream/node. For example, $S_1(e_{11})$-$S_2(e_{21})$ is an event tree for the streams of Figure 2, where the event e_{11} (resp. e_{21}) is selected from S_1 (resp. S_2). An event tree T is also characterized by its start and end time, *T.start* and *T.end*, respectively, which are defined as follows: $T.start = min_t \{e_{ij}.t \mid e_{ij}\in T\}$ and $T.end = max_t \{e_{ij}.t \mid e_{ij}\in T\}$.

The score of an event tree T is

$$\text{score(T)} = \frac{1}{\sum_{edge\ (S_x,S_y)\in T} \frac{1}{a(S_x,S_y)}} \tag{1}$$

which ensures that larger trees receive smaller score. If we would take the sum of the weights instead, then the score of an event tree would increase as the event tree gets bigger which is counterintuitive, since smaller event trees typically correspond to tighter association. For example, the score of $S_1(e_{11})$-$S_2(e_{21})$ is 0.2. Notice that the score of an event tree only depends on the corresponding streams and not on the particular events.

Continuous Keyword Query

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 $q.keywords:=(kw_1,...,kw_m)\in (A^*)^N$, a time window length *q.matchingWindow*, and a real number *q.associationThreshold* .

The answer of a query q on a set $\boldsymbol{S}$ of text streams is a sequence of all event trees T with 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) and still have all keywords contained, that is, there is no leaf event $e \in T$ such that $keywords(e) \subset keywords(T\text{-}e)$, and
3. the events in T occur within a window of time length *q.matchingWindow*, that is, $T.end\text{-}T.start \leq q.matchingWindow$, 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 in Figure 3. 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.

For instance, the query in Figure 2 has the following answer: $S_2(e_{21},e_{22})$, $S_1(e_{11})$-$S_2(e_{22})$. The second result would be missed by a traditional alert system. Also note that $S_1(e_{11})$-$S_3(e_{31})$ is not an answer because streams S_1 and S_3 are not correlated (see Figure 3).

4. CHALLENGES AND OVERVIEW OF OUR APPROACH

Given the previous work in keyword proximity search [GSVG98, BNH+02, ACD02, HP02], 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 $[t_{now}\text{-}associationWindow, t_{now}]$.
- 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.

[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.

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.

5. INCREMENTALLY COMPUTE 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 e_{QR} has the following property: $q.keywords \cap e_{QR}.content \neq \varnothing$. Therefore, they characterize events containing at least one of the query keywords. For example, if *q.matchingWindow=1h* 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.

```
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)
14.    If T not null then output event tree T
     }
   }
```

Figure 4: Tree Algorithm.

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 1.

The tree algorithm is described in Figures 4 and 5 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.

```
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 of text streams V 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
8.   return null
```

Figure 5: Event tree computation algorithm.

Notice how in Line 4 of Figure 4 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.

Example 3: Consider query q *with q.keywords:= (Florida, mortgage), 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* = “<u>Florida</u> real estate is hot.”

e_2: in stream S_2, e_2*.content* = “I live in <u>Florida</u>.”

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

e_3: in stream S_3, e_3*.content* = “More people apply for <u>mortgage</u>.”

e_4: in stream S_1, e_4*.content* = “<u>Florida</u> is growing.”

e_5: in stream S_2, e_5*.content* = “People move to <u>Florida</u>.”

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 6 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 4.
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 5 are simply single edges since only two stream nodes are in graph G_T for any candidate result.□

6. EXPERIMENTS

In our experiments we used the Enron emails dataset (see [KY04] for a description). 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 the body. (The email subject could also be viewed as part of the description.) Therefore, now an email thread corresponds to a stream of text.

In order to evaluate the performance of the Tree algorithm, we separate the execution of the algorithm from the edge weights calculation. That is, the times reported in this section do not include the time for the graph maintenance, which for this experiment are already precomputed. 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*.

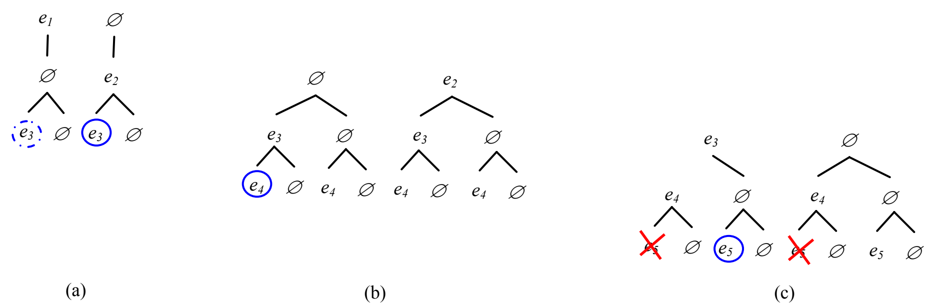

Figure 6: Snapshots of forest *C* in Tree algorithm.

We measure the query execution time per query-related event. We utilize two keywords per query and test the performance of the system on 50 continuous keyword queries. In Figure 7 we report the *median* 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 a extensive query time range.

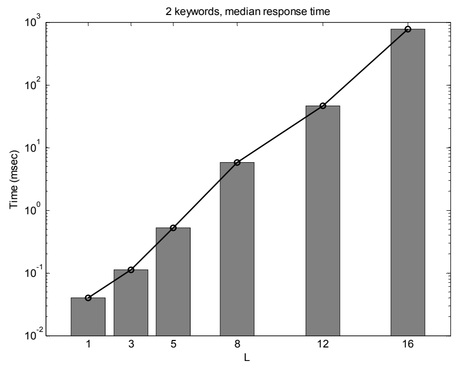

Figure 7: Median execution times for varying L.

7. REFERENCES

[ABB+02] Arvind Arasu, Brian Babcock, Shivnath Babu, Jon McAlister, Jennifer Widom: Characterizing Memory Requirements for Queries over Continuous Data Streams. PODS 2002

[ACD02] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. ICDE, 2002

[AN04] Ahmed Ayad, Jeffrey F. Naughton: Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. SIGMOD 2004

[BHP04] A. Balmin, V. Hristidis, Y. Papakonstantinou: Authority-Based Keyword Queries in Databases using ObjectRank. VLDB, 2004

[BNH+02] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti and S,Sudarshan: Keyword Searching and Browsing in Databases using BANKS. ICDE, 2002

[CGMR05] Graham Cormode, Minos N. Garofalakis, S. Muthukrishnan, Rajeev Rastogi: Holistic Aggregates in a Networked World: Distributed Tracking of Approximate Quantiles. SIGMOD 2005

[FCFP05] Cristian Fiorentino, Mariano Cilia, Ludger Fiege, Alejandro P. Buchmann: Building a Configurable Publish/Subscribe Notification Service. DAIS 2005

[FJL+01] 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

[FLN01] R. Fagin, A. Lotem, M. Naor. Optimal aggregation algorithms for middleware. PODS, 2001

[GGR03] Sumit Ganguly, Minos N. Garofalakis, Rajeev Rastogi: Processing Set Expressions over Continuous Update Streams. SIGMOD 2003

[GSVG98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina: Proximity Search in Databases. VLDB, 1998

[HGP03] V. Hristidis, L. Gravano, Y. Papakonstantinou: Efficient IR-Style Keyword Search over Relational Databases. VLDB, 2003

[HP02] V. Hristidis, Y. Papakonstantinou: DISCOVER: Keyword Search in Relational Databases. VLDB, 2002

[HVVY06] Vagelis Hristidis, Oscar Valdivia, Michalis Vlachos, Philip S. Yu: Continuous Keyword Search on Multiple Text Streams. Poster paper, ACM CIKM 2006

[IAE03] Ihab F. Ilyas, Walid G. Aref, Ahmed K. Elmagarmid: Supporting Top-k Join Queries in Relational Databases. VLDB 2003

[KY04] B. Klimt and Y. Yang. Introducing the Enron corpus. First Conference on Email and Anti-Spam (CEAS), 2004

[RSS05] RSS 2.0 Specification. http://blogs.law.harvard.edu/tech/rss, 2005

[ZKOS05] Rui Zhang, Nick Koudas, Beng Chin Ooi, Divesh Srivastava: Multiple Aggregations Over Data Streams. SIGMOD 2005