Category-Based Infidelity Bounded Queries over Unstructured Data Streams

Manish Bhide and Krithi Ramamritham, *Fellow*, *IEEE*

Abstract—We present the *Caicos* system that supports continuous infidelity bounded queries over a data stream, where each data item (of the stream) belongs to multiple categories. *Caicos* is made up of four primitives: Keywords, Queries, Data items, and Categories. A Category is a virtual entity consisting of all those data items that belong to it. The membership of a data item to a category is decided by evaluating a Boolean predicate (associated with each category) over the data item. Each data item and query in turn are associated with multiple keywords. Given a keyword query, unlike conventional unstructured data querying techniques that return the top-K documents, *Caicos* returns the top-K *categories* with infidelity less than the user specified infidelity bound. *Caicos* is designed to continuously track the evolving information present in a highly dynamic data stream. It, hence, computes the relevance of a category to the continuous keyword query using the data items occurring in the stream in the recent past (i.e., within the current "*window*"). To efficiently provide up-to-date answers to the continuous queries, *Caicos* needs to maintain the required metadata accurately. This requires addressing two subproblems: 1) Identifying the "*right*" metadata that needs to be updated for providing accurate results and 2) updating the metadata in an efficient manner. We show that the problem of identifying the right metadata can be further broken down into two subparts. We model the first subpart as an inequality constrained minimization problem and propose an innovative iterative algorithm for the same. The second subpart requires us to build an efficient dynamic programming-based algorithm, which helps us to find the right metadata that needs to be updated. Updating the metadata on multiple processors is a scheduling problem whose complexity is exponential in the length of the input. An approximate multiprocessor scheduling algorithm is, hence, proposed. Experimental evaluation of *Caicos* using real-world dynamic data shows that *Caicos* is able to provide fidelity close to 100 percent using 45 percent less resources than the techniques proposed in the literature. This ability of *Caicos* to work accurately and efficiently even in scenarios with high data arrival rates makes it suitable for data intensive application domains.

Index Terms—Continuous query, data stream, category-based query, threshold queries

1 Introduction

1.1 Problem Landscape

Our problem landscape (Fig. 1) consists of four primitives: Keywords, Queries, Data items, and Categories. The input stream consists of data items each of which can belong to one or more categories. Each data item as well as a query is made up of multiple keywords.

Given a query, a regular search engine (Fig. 1a) returns individual data items. A window-based search engine on the other hand (Fig. 1b) returns the data items present in the most recent window. A category-based search engine (Fig. 1c) returns categories that are virtual entities consisting of multiple data items. *Caicos*, which is the subject of this paper, is a category-based infidelity bounded (windowed) continuous query engine (Fig. 1d). It builds on top of all the three preceding concepts and (continuously) returns the top-K categories, where the categories are made up of data items present in a moving window.

• M. Bhide is with IBM Software Lab, Mindspace 3A, Madhapur, Hyderabad, India. E-mail: abmanish@in.ibm.com.
• K. Ramamritham is with the Department of Computer Science and Engineering, IIT Bombay, Powai, Mumbai 400076, India.
E-mail: krithi@iitb.ac.in.

Manuscript received 30 Dec. 2011; revised 18 June 2012; accepted 1 Oct. 2012; published online 11 Oct. 2012.
Recommended for acceptance by B.C. Ooi.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2011-12-0809.
Digital Object Identifier no. 10.1109/TKDE.2012.200.

1.2 Motivation 1

Consider a scenario where a major automobile company, Mac, has recently launched an advertisement campaign for their next generation sedan named *Hestia*. The brand manager of the sedan wants to track the evolving opinion/reaction of potential customers as different phases of the campaign are rolled out. This information is available in the form of forum postings, wiki entries, blog posts, and so on, which are continuously generated on the web forming a data stream. The only mechanism available to the brand manager today, is to fire a keyword query "Mac Hestia sedan" on a category-based search engine (working over the data stream) [4] so as to find the blog posts, forum postings, wiki entries, and so on, which are relevant to the sedan. Unlike conventional search engines which report the top-K documents, a category-based search engine reports the top-K categories related to the keyword query. Example categories in this scenario may include: postings about high maintenance cars, postings about fuel efficiency problems, and so on. Thus, the advantage of using a category-based search engine is that it enables the brand manager to quickly understand the key categories/issues being discussed by its consumers without going through thousands of search results.

However, the major drawback of these search engines is that they do not report the evolving opinion of the consumers—something that is required by the brand manager to assess the effect of the marketing campaign.

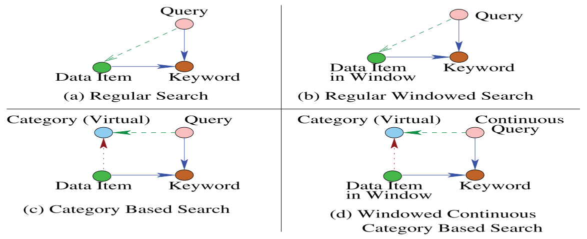


Fig. 1. Problem landscape.

One could argue that the evolving opinion can be captured by querying the search engine periodically and finding the difference in the search results. However, notice that the category-based search engines available today [4] report the relevant categories with respect to the contents of the entire web. Given the size of the web, the categories that have recently become relevant due to the newly added blog posts and forum postings will be overshadowed by the older documents/categories. For example, in the brand manager scenario, people could be posting about the fuel efficiency issues due to the increased engine size of the new sedan. The category-based search engines will not report categories related to the *fuel efficiency issues* at the top, as these postings will be overshadowed by the large number of other posts that already exist on the web. Thus, by the time the brand manager knows about this issue, irreparable damage could have already been done in the minds of prospective customers. Another point highlighted by this scenario is that the brand manager might not care about getting 0 percent infidelity in the reported top-K results, but would be satisfied as long as at least one of the categories related to the fuel efficiency issue is reported in the top-K.

1.3 Motivation 2

Consider another example from the financial domain where a stock analyst wants to continuously track the profile/category of people buying IBM stock. The data stream consists of the stock transactions that can be categorized based on the stock(s) associated with the transactions, buyer/seller profiles, and so on. Sample categories could include, “Transactions made by investors investing in green companies,” “Transactions made by high value customers,” and so on. The analyst can register a continuous query “IBM” with *Caicos* and would be continuously notified of the top-K categories/profiles. It could happen that *Caicos* suddenly lists categories such as 1) “Investors investing in green companies,” 2) “Investors listing clean energy as their interest (in their profile),” and 3) “Investors dealing with carbon credit” in the top-10 ($K = 10$) results. A little investigation by the analyst might reveal that IBM has made an announcement in a conference that it has made significant strides in desalinating sea water. Notice that, as in the previous scenario, this problem cannot be solved by using conventional search techniques as the category about green investors will be overshadowed by the thousands of other transactions of IBM. Another crucial point is that if the results have 0 percent infidelity, then all the three categories about green companies (listed above) would be present in the top-10 results. However, even if one of the above three categories is listed in the top-10 results, the analyst would still have done further investigation. Thus, the analyst would be fine with 20 percent infidelity in the results.

The two scenarios show that there is a need for a technique which

- supports continuous queries,
- finds the top-K categories based only on recently streamed information, and
- reports the top-K with a bounded infidelity (as compared to the actual result).

As the system is evolving, the top-K reported by the system will differ from that reported by an Oracle. The *infidelity* of the top-K *categories* reported by a system (such as *Caicos*) at a given instant is the number of categories that are incorrectly reported as being in the top-K. On the other hand, the system is said to *violate the infidelity bound* if the infidelity in the top-K is more than the user specified infidelity bound ($\mathcal{F}$: 20 percent in above example). If the $\mathcal{K}$(q) and $\mathcal{K}^{\mathcal{H}}$(q) are the top-$K$ categories reported by *Caicos* and the actual top-K, respectively, then *Caicos* system is said to violate the infidelity bound if the following equation is not satisfied:

$$\frac{|\mathcal{K}^{\mathcal{H}}(q) - \mathcal{K}(q)|}{K} \leq \mathcal{F}. \tag{1}$$

The infidelity of *Caicos* for the query q over the query life time, i.e., $Infidelity(q)$ is the percentage time duration for which (1) does not hold true. Extending this further, the infidelity of *Caicos* is the average of infidelity for all the continuous queries ($\mathcal{Q}$) registered in the system:

$$Infidelity(\mathcal{Q}) = \sum_{i=1}^{|\mathcal{Q}|} \frac{Infidelity(q_i)}{|\mathcal{Q}|}. \tag{2}$$

In this paper, we propose the *Caicos* system that continuously reports the top-K categories (selected from a predefined list of categories) relevant to a keyword query, where the categories are made up of a subset of the most recent data items available in the data stream (i.e., current “window”). For example, in the first scenario, the relevance of the category “*Blog posts about fuel efficient cars*” will be computed with respect to the data items belonging to the category amongst those added in the recent past. Notice that *Caicos* will also be useful for building reputation monitoring dashboards for companies interested in tracking the reputation of their products and services on the web. As shown by the two motivating scenarios, companies would not be interested in finding the exact top-K, but would be satisfied as long as the reported top-K are the true top-K within a tolerable bound. Hence, the *Caicos* system reports the top-K categories with a bounded infidelity, i.e., if the user specified infidelity bound is 20 percent, then it implies that the user is willing to tolerate an infidelity of 20 percent in the reported top-K results. This bound will vary based on the usage scenario and would be deduced by the user based on the quality of the results.

To compute the top-K categories efficiently, *Caicos* maintains *metadata* consisting of, among other things, information about the set of data items belonging to a category, data items in the current window, and so on (more details in Section 2.3). Notice that for computing the metadata accurately, *Caicos* first needs to (dynamically) find the set of categories corresponding to each data item in the stream. This process of finding whether a data item belongs to a particular category and updating the metadata is called *refreshing a data item-category pair*. In our first motivating example, this process will require the execution of a text classifier whereas in the second example this would involve the execution of multiple expensive SQL joins to analyze the current portfolio of the user. Thus, given a category "Blog posts about hybrid cars," a text classifier will take a blog post as input and decide whether it is about hybrid cars. Similarly, for the category "Transactions made by investors investing in Green companies," the SQL queries will involve multiple joins to find the composition of the users' portfolio.

The main challenge that *Caicos* is designed to address arises due to the high cost associated with refreshing a data item-category pair. If the text classifier in the first example or the multiple SQL joins in the second motivating example takes 10 milliseconds to execute, then with 1,500 categories, we will require 15 seconds to find all the categories of a single data item. But, the rate of arrival of new data items will be much higher than this; as according to an estimate around 13 blog posts are generated per second [17] and the number of transactions in the stock market is also very high. Hence, by the time we finish processing one data item, many new data items would have been added. Thus, the metadata captured by a *Uniform Refresh Technique* will become increasingly stale leading to an infidelity in the top-K results.

Hence, what is needed is a real time system that intelligently refreshes the metadata corresponding to a select set of categories using a select set of data items. Incorporating this idea, we have developed the *selective update technique*, which is used by *Caicos*. The selective update technique uses the infidelity bound specified by the user to find the subset of the data item-category pairs, which can be refreshed such that they provide the maximum reduction in infidelity of the registered continuous queries. This way, we utilize the processing time effectively, by ignoring the unimportant data item-category pairs, and the saved time can then be used for processing the remaining data updates in a timely manner. Realizing this desirable property, however, requires several research issues to be tackled. In this paper, we describe the details of *Caicos* that incorporates our novel solutions to these issues. Specifically, we make the following research contributions:

- *Caicos* is the very first attempt to formulate and present an efficient solution to the problem of category-based infidelity bounded windowed continuous queries over streaming data.
- The infidelity bound specified by the user is in terms of the top-K, i.e., how many of the reported top-K categories should actually be in top-K. We convert this bound to an objective function using a constraint optimization-based technique so that we can achieve the least possible violations of the infidelity bound with minimum number of refreshes.
- A data item in *Caicos* can affect the relevance of a category to a keyword query (which in turn affects the infidelity) if and only if it belongs to that category—a fact that is known only after executing the text classifier or executing the SQL queries. However, in the interest of efficiency it is imperative to find the "*right*" set of data item-category pairs without running the expensive text classifiers or SQL joins. We, hence, propose an elegant algorithm that helps us to estimate the relevance without executing the text classifiers.
- Finding the correct data items and categories to refresh such that we get the least possible violations of the infidelity bound with minimum number of refreshes can be a very challenging task. We present a dynamic programming-based algorithm that helps us to identify the best set of data item category pairs to be refreshed in view of the registered continuous queries.
- To satisfy the real time constraints, *Caicos* needs a refresh strategy such that the refresh of the selected data item-category pairs finishes in the shortest possible time interval and adapts to the ever changing data and category characteristics of the stream. We, hence, devise a feedback-based mechanism to continuously adjust the refresh process to the dynamics of the input stream. We further show that the refresh process maps to a single-stage scheduling problem, whose complexity is exponential in the length of the input. *Caicos*, therefore, uses a polynomial time approximate multiprocessor scheduling algorithm for performing the updates in parallel.
- *Caicos*'s design makes very efficient use of the available processing power and, hence, can be applied to internet scale problem spaces in a scalable manner. Experimental evaluation of *Caicos* using real-world data shows that *Caicos* is able to maintain an fidelity close to 100 percent using 45 percent less resources than the uniform refresh technique.

2 Preliminaries

2.1 Data Model

Consider a continuously evolving stream of data items $d_1, d_2, d_3, \ldots$ Each data item d is associated with a set of attributes $A(d)$ and a multiset of keywords $E(d)$ drawn from a universe of terms $\mathcal{E}$. Each data item belongs to one or more categories from a set of categories $\mathcal{C}$. Each category c is associated with a Boolean predicate $p_c(\cdot)$ that takes as input a data item d and tells whether d belongs to category c. $p_c(\cdot)$ is evaluated over $A(d)$ and $E(d)$. Given a keyword query, at a high level, the goal is to find the top-K categories relevant to the keyword query with an infidelity less than the user specified infidelity bound $\mathcal{F}$. The relevance of a category c to a keyword query is computed based on the relevance of those data items (whose $p_c(\cdot)$ is true) to the keyword query.

In the real world, the above mentioned data items (*dynamic data*) would map to blog posts, forum postings, financial transactions, and so on. The keywords $E(d)$ for a blog post would be the text of the posting and $A(d)$ would be the attributes of the blog authors profile. Similarly, $E(d)$

for the financial transaction would be the text of the company profile involved in the transaction and $A(d)$ would be the attributes of the user executing the transaction. Examples of categories (which could be predefined or added dynamically to the category-based search system) include: "Blog posts about hybrid cars," "Transactions made by investors investing in Green companies," and so on. As mentioned earlier, the predicate for the first category would be realized by a text classifier, which would take the posting as input and decide whether the posting is related to hybrid cars whereas that for the second category would involve execution of SQL queries to analyze the current portfolio of the user.

In this paper, for ease of explanation, we do not measure time in absolute terms, but in terms of *time step*. Updates to the stream with one or more data items causes the time step to be incremented proportionately (i.e., equal to the number of data items added). Thus, there is a one-to-one mapping between a time step and the data item added to the stream in that time step. The window (of size $\mathcal{W}$) is also measured in terms of time step.[1] Let $D_s(q)$ denote the set of data items present in the window of query q at time-step s, then

$$D_s(q) = \{d_i : (s - \mathcal{W} + 1) \leq i \leq s\}.$$

The data set ($M_s(c)$) of a category c, at time-step s, is defined to be the set of all data items (d) present in the current window at time-step s that map to c (i.e., $p_c(d) = 1$):

$$M_s(c) = \{d \in D_s : p_c(d) = 1\}.$$

2.2 Queries

Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_{|\mathcal{Q}|}\}$ be the set of continuous queries registered in the system. Each (user defined) continuous query $q \in \mathcal{Q}$ consists of a set of terms $\{t_1, t_2, \ldots, t_\ell\}$ and a window size $\mathcal{W}(q)$. Query q is also associated with a begin and end time (denoted by $S(q)$ and $F(q)$, respectively). Given $\mathcal{Q}$, D_s, and the set of categories $\mathcal{C}$, our goal is to find the set of categories most relevant to each $q \in \mathcal{Q}$ at each time step $S(q) \leq s \leq F(q)$ with an infidelity less than the specified bound $\mathcal{F}$. We need to evaluate the keyword query at time-step s with respect to the data items present in the window ending at that time step, i.e., $D_s(q) = \{d_i : (s - \mathcal{W}(q) + 1) \leq i \leq s\}$.

The top-K categories are found using a scoring function. Let $\mathcal{I}$ be the function that takes as input a category c and a keyword t and computes the Score of the category with respect to t. Further, let $\mathcal{G}$ be a function that combines the $\mathcal{I}(c,t)$ values for each $t \in q$ to arrive at $\text{Score}(c,q)$. Let the score of a category c with respect to a keyword query q be represented by $\text{Score}(c,q)$. This would depend on the score of c for each keyword present in q. Then,

$$\text{Score}(c,q) = \mathcal{G}(\mathcal{I}(c,t_1), \mathcal{I}(c,t_2), \ldots, \mathcal{I}(c,t_\ell)). \quad (3)$$

Given a continuous query q, the goal is to continuously report the categories having the top-K scores (K is an input parameter) among all the categories defined in the system with an infidelity less than the user specified infidelity bound $\mathcal{F}$.

1. A window represented in absolute time can be converted into time steps by using the average rate of arrival of data items.

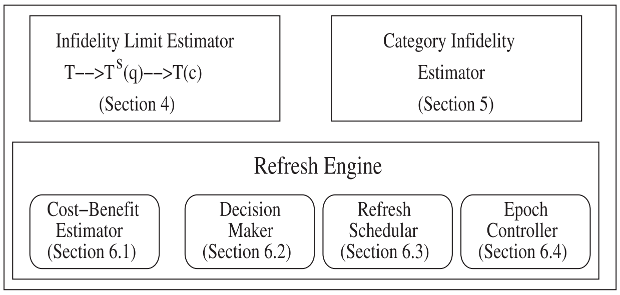

Infidelity Limit Estimator
T-->T^S(q)-->T(c)
(Section 4)
Category Infidelity Estimator
(Section 5)
Refresh Engine
Cost-Benefit Estimator (Section 6.1)
Decision Maker (Section 6.2)
Refresh Schedular (Section 6.3)
Epoch Controller (Section 6.4)

Fig. 2. *Caicos* architecture.

2.3 Metadata

As mentioned earlier, the query answering process makes use of *metadata*, which helps it to efficiently find the value of $\text{Score}(c,q)$ for each c. The metadata includes 1) information about $\mathcal{I}(c,t)$ values (with respect to $D_s(q)$) for all $c \in \mathcal{C}$ and $t \in \mathcal{E}$, 2) information needed to compute the $\mathcal{G}$ function, and 3) data structures such as inverted indexes that help to speed up the query answering process. The inverted index maps each keyword t, to the set of all categories whose data set contains a data item having t. This metadata is bound to change as $D_s(q)$ and, hence, $M_s(c)$ changes with the passage of time s. This implies that to compute the Score accurately, we must keep the metadata up-to-date as new data are added and the old data move out of the current window. If data arrival rate is high, this is a challenging task—a challenge that *Caicos* is designed to address.

It is important to note that the predicate associated with each category is domain dependent and is, hence, provided as input to *Caicos*. It is also important to note that in the first motivating example the time overhead associated with the evaluation of the text classifiers cannot be avoided by building multiclass classifiers. Studies [1] have shown that for a large number of categories, building such classifiers or a hierarchical set of classifiers does not work due to an explosion in the number of class labels leading to very poor results.

3 Caicos System Overview

The *Caicos* system (Fig. 2) consists of three major components:

- Infidelity Limit Estimator
- Category Infidelity Estimator
- Refresh Engine

Caicos uses the selective update technique to intelligently refreshes the metadata corresponding to a select set of categories using a select set of data items. To implement this strategy *Caicos* needs to 1) find the set of data item and categories that have to be refreshed and 2) refresh them on multiple processors. As mentioned in Section 2, *Caicos* computes the Score for each category using the available metadata. Thus, to find the set of data item and categories that need to be refreshed, we need to define a decision function based on the Score value. However, the input to *Caicos* is the infidelity bound $\mathcal{F}$, which is represented in terms of number of categories. Hence, there needs to be correlation between the $\mathcal{F}$ value and the decision function. This correlation (see Fig. 3 and Table 1) is achieved by

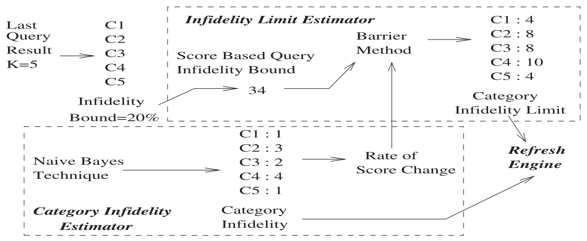


Fig. 3. Solution components.

computing two values 1) Infidelity limit (computed by the Infidelity Limit Estimator) and 2) Category Infidelity (computed by the Category Infidelity Estimator). Thus, the first task of finding the data item and categories that are to be refreshed is done by the *Infidelity Limit Estimator* and the *Category Infidelity Estimator*, whereas the second task is done by the *Refresh Engine*.

The *Infidelity Limit Estimator* is responsible for finding the maximum permissible infidelity in the Score of a category. This is done by converting the infidelity bound $\mathcal{F}$ specified by the user into an infidelity limit $\mathcal{F}(c)$ for each category c relevant to the registered continuous queries. For example, if the user specified infidelity bound is 20 percent, then the *Infidelity Limit estimator* could estimate the infidelity limit of say, 2 units for a category c_1 and 3 units for a category c_2. The *Category Infidelity Estimator* on the other hand estimates the infidelity in each category based on the data items present in the current window. For example, if the Score of a category c_1 at time s was 2 units and it is currently estimated to be 5 units, then the estimated infidelity of c_1 would be 3 units.

The Infidelity limit estimator computes the infidelity limit such that the minimum number of categories exceed their infidelity (which is computed by Category Infidelity estimator). It achieves this by formulating an inequality constrained minimization problem, which it solves using the Barrier method (as explained in Section 4). The Category Infidelity estimator, on the other hand, has to estimate the category infidelity in an efficient manner. The category infidelity can be computed by executing the Boolean predicate associated with each category. However, given the high rate of arrival of data items, the Category Infidelity Estimator computes the infidelity without executing the expensive Boolean predicate as explained in Section 5.

Once the infidelity limit and the estimated infidelity of a category are known the next task is to 1) find the set of data item-category pairs that need to be refreshed and 2) refresh them on the available processors. This task is done by the *Refresh Engine*. As is intuitive, the Refresh Engine refreshes only those categories whose estimated infidelity has exceeded their infidelity limit (which is the decision function). However this, as we show in Section 6, is a very complex task especially in the presence of multiple continuous queries. We, hence, propose a dynamic programming-based solution for finding the best set of data item-category pairs to be refreshed. We thereafter show that the problem of finding an optimal schedule for refreshing the data item-category pairs on multiple processors maps to the single-stage scheduling problem whose complexity is exponential in the length of the input. We, hence, use an approximate algorithm for finding a refresh schedule (details in Section 6.3). *Caicos* works in an iterative manner by dividing time into epochs. During each epoch, it finds the best set of data item-category pairs to refresh and then refreshes them as per the schedule given by the Refresh Scheduler.

TABLE 1
Summary of Concepts

Concept	Meaning
Query Infidelity Bound	Maximum number of incorrect results tolerable in top-K
Score based Query Infidelity Bound	Infidelity bound measured in terms of Score
Infidelity Limit	Maximum permissible infidelity in Score of a category
Category Infidelity	Difference in last know Score and current estimated Score for a category

Before getting into the details of *Caicos*, we first provide an overview of the scoring function in the next section.

3.1 Scoring Function

In this paper, we have used the most widely used scoring technique of information retrieval, which is based on term frequencies (tf) and inverse document frequency (idf) [16].

Term Frequency. The *term-frequency* of a term t in a category c at time-step s is defined to be the number of times t occurs in the data set of c normalized by the total number of terms present in it. The size normalization helps to avoid the bias toward categories with large number of terms. Let $f(d,t)$ denote the number of times t appears in d. Then,

$$tf_s(c,t) = \frac{\sum_{d:d\in M_s(c)} f(d,t)}{\sum_{t'\in\mathcal{E}}(\sum_{d:d\in M_s(c)} f(d,t'))}. \quad (4)$$

Inverse Document Frequency. The inverse document frequency seeks to scale down the effect of those terms that occur more frequently across different categories. At a time-step s, the idf of a term t is calculated as follows:

$$idf_s(t) = 1 + log\left(\frac{|\mathcal{C}|}{|\mathcal{C}'|}\right), \quad (5)$$

where $\mathcal{C}$ is the set of all categories in the system and $\mathcal{C}'$ is the set of categories whose data set contain t, i.e., $\exists d \in M_s(c)[f(d,t) \geq 1]$.

Score. For a keyword query $q = \{t_1, t_2, \ldots, t_\ell\}$, the Score of a category c at time-step s is given by

$$\text{Score}_s(c,q) = \sum_{t_i\in q}(tf_s(c,t_i) \times idf_s(t_i)). \quad (6)$$

One shortcoming of the above scoring function is that it does not factor in the distance between the keywords. Thus, for a query $q = \{t_1, t_2\}$, a category c_1 which has both keywords appearing close to each other would have the same Score as that of a category c_2 in which they appear far from each other. We address this shortcoming by using the concept of *proximity*, which is similar to that used by Google. Google classifies the distance between keywords into 10 bins based on whether the keywords are in a single phrase (bin $= 1$) or are "not even close" (bin $= 10$). Given that ℓ is the number of keywords in the query, our modified scoring function can be stated as

$$\text{Score}_s(c,q) = \mathcal{D}(c,q) \times \sum_{t_i \in q} (tf_s(c,t_i) \times idf_s(t_i))$$
$$\mathcal{D}(c,q) = \frac{2}{\ell(\ell-1)} \times \sum_{j=1}^{\ell-1} \sum_{k=j+1}^{\ell} \frac{1}{\mathcal{B}(c,t_j,t_k)}, \tag{7}$$

where $\mathcal{B}(c,t_j,t_k)$ gives the bin number corresponding to the minimum distance between the keywords t_j and t_k in the category c. Consider an example of a keyword query consisting of three keywords $q = \{t_1,t_2,t_3\}$. Let the three keywords appear as consecutive words in a category c_1. Then, the value of $\mathcal{D}(c_1,q)$ would be

$$\mathcal{D}(c_1,q) = \frac{2}{3\times 2} \times \left(\frac{1}{\mathcal{B}(c_1,t_1,t_2)} + \frac{1}{\mathcal{B}(c_1,t_1,t_3)} + \frac{1}{\mathcal{B}(c_1,t_2,t_3)}\right).$$

As the keywords appear consecutively, the value of $\mathcal{B}(c_1,t_1,t_2)$, $\mathcal{B}(c_1,t_1,t_3)$, and $\mathcal{B}(c_1,t_2,t_3)$ would all be 1. Hence, the value of $\mathcal{D}(c_1,q)$ would be 1. Now consider the case of a category c_2 where t_1 and t_2 appear consecutively but t_3 is far away from t_1 and t_2. In such a case, the value of $\mathcal{B}(c_1,t_1,t_3)$ and $\mathcal{B}(c_1,t_2,t_3)$ would be larger say 4, whereas the value of $\mathcal{B}(c_1,t_1,t_2)$ would be 1. Hence, the value of $\mathcal{D}(c_1,q)$ would be: $\frac{2}{3\times 2} \times (1 + \frac{1}{4} + \frac{1}{4}) = 0.5$, which will help to reduce the value of $\text{Score}(c_2,q)$.

Thus, $\mathcal{D}(c,q)$ factors in the distance between the query keywords in the category c and $\frac{\ell(\ell-1)}{2}$ helps to normalize the value of $\mathcal{D}(c,q)$. Hence, $\mathcal{D}$ ensures that a category which has the query keywords in close proximity of each other will have a higher Score as compared to others. For this $tf \cdot idf$-based scoring function, the metadata that needs to be computed for finding the Score includes information about tf, idf, and $\mathcal{D}$ with respect to all keywords, categories, and continuous queries.

It is important to note that *Caicos* can easily make use of other scoring functions such as BM25 [15]. The BM25 scoring function also consists of tf and idf components and can be used without requiring any changes to *Caicos*. More details about this scoring function are provided in Section 7.2.3.

4 Infidelity Limit Estimator

The goal of the *Infidelity Limit Estimator* is to find the infidelity limit $\mathcal{F}(c)$ for each category c. The key challenge in finding the infidelity limit of a category is the fact that the query infidelity bound is specified in terms of top-K, whereas the infidelity limit of a category has to be represented in terms of the $tf \cdot idf$ score, i.e., the unit of infidelity limit of a category and query infidelity bound are different. The *Infidelity Limit Estimator* addresses this challenge by converting the query infidelity bound represented in terms of top-K into a *Score-Based Query Infidelity Bound* $\mathcal{F}^s(q)$, which is represented in terms of the $tf \cdot idf$ score. Once the score-based query infidelity bound has been computed, we then need to find the infidelity limit for all the categories related to the continuous query such that the following equation is satisfied:

$$\mathcal{F}(c_1) + \mathcal{F}(c_2) + \cdots + \mathcal{F}(c_n) = \mathcal{F}^s(q), \tag{8}$$

where $c_1, c_2, \ldots, c_n$ are the categories related to a continuous query q (details of what we mean by "related" will be clarified in Section 4.2). We first present our procedure to find the score-based query infidelity bound in the next section and then present the procedure of finding the infidelity limit for a category in Section 4.2.

4.1 Finding Score-Based Query Infidelity Bound

The *Category Infidelity Estimator* measures the infidelity in the category in terms of its $tf \cdot idf$ score. In other words, it measures the difference in the last reported $tf \cdot idf$ score for a category with its (estimated) score at a given instance. Hence, if we can convert the query infidelity bound to a score-based query infidelity bound, then it can be easily compared against the category infidelity estimated by the *Category Infidelity Estimator*. However, this is a challenging task as the query infidelity bound is computed in terms of position in top-K, whereas the category infidelity is measured in terms of the $tf \cdot idf$ score. For example, the query infidelity bound could be 20 percent, which implies that the maximum number of categories incorrectly reported as being in top-K can be 20 percent. Our goal is to convert this 20 percent into a $tf \cdot idf$ value say, 34 units so that it can be compared with the category infidelity estimated by the *Category Infidelity Estimator*. Thus, as one can notice, because the units itself are different, it is not possible to devise a theoretical approach to do this conversion. Hence, *Caicos* uses an intelligent strategy where it substitutes the score-based query infidelity bound by its lower bound. We explain next the technique used for computing this lower bound.

Let $c_1, c_2, \ldots, c_{15}$ be the top 15 categories ordered based on their $tf \cdot idf$ scores for a continuous query q. Let the user specified query infidelity bound $\mathcal{F}(q)$ be 30 percent and let $K = 10$. If the query infidelity bound is to be violated with the least possible change in the $tf \cdot idf$ scores, then it can happen in a scenario where the categories closest to the Kth position which are inside the top-K (i.e., c_8, c_9 and c_{10}) move out and those closest to the Kth position but outside the top-K (i.e., c_{11}, c_{12}, and c_{13}) move in. In this scenario, the sum total of the change in the $tf \cdot idf$ scores of the six categories represents the minimum change that needs to happen in the $tf \cdot idf$ scores of the categories (related to the query) for the query infidelity bound to be violated. This is nothing but the lower bound on the score-based query infidelity bound. *Caicos* uses this lower bound as the score-based query infidelity bound. This might seem to be overtly restrictive, but in reality (as we explain in Section 6) this helps us to achieve good results.

Thus, based on the above discussion, the score-based query infidelity bound can be represented as

$$\mathcal{F}^s(q) = \sum_{i \in [k-\mathcal{F}(q),\, k+\mathcal{F}(q)]} \left| \text{Score}(c_i,q) - \frac{\text{Score}(c_k,q) - \text{Score}(c_{k+1},q)}{2} \right|. \tag{9}$$

Once the score-based query infidelity bound has been computed, the next task is that of finding the infidelity limit of a category.

4.2 Finding Infidelity Limit

Caicos computes the infidelity limit of a category such that the number of data item-category pairs that need to be refreshed (for maintaining the user specified query infidelity bound) is minimized. As mentioned earlier, the *Refresh Engine* refreshes only those categories whose estimated infidelity has exceeded its infidelity limit. We use this fact to ensure that the infidelity limits of categories are such that they reduce the number of data item-category pairs that need to be refreshed. Notice that if we know the rate of change of Score of a category c_i (represented by r_i), then the time at which a category needs to be refreshed (TTR) can be represented as

$$TTR = \frac{\mathcal{F}(c_i)}{r_i}. \tag{10}$$

We use this fact to formulate the problem of determining the infidelity limit of a category as an inequality constrained minimization problem. Assuming that the value of r_i is accurate (and we refresh a category using (10)), if the value of infidelity limit of a category is computed such that it satisfies (8), then we would get 0 percent infidelity (i.e., no violation of the infidelity bound). Thus, we need to compute the value of the infidelity limit for each category such that (8) is not violated. Using (10), the number of refreshes per second can be calculated as

$$Refreshes\ Per\ Second = \sum_{c_i \in \mathcal{K}(q)} \left(\frac{|r_i|}{\mathcal{F}(c_i)} \right), \tag{11}$$

where $\mathcal{K}$ in the above formula represent the top-$(K + \frac{K}{2})$ categories related to the query q. This utility of $K + \frac{K}{2}$ categories has been found out empirically (details in Section 5.2).

Our aim of minimizing the number of refreshes can be achieved by finding $\mathcal{F}(c)$ values such that 1) (11) is minimized and 2) (8) is not violated. This can be done using convex optimization techniques (notice that (11) can be trivially proved to be convex for $\mathcal{F}(c_i) > 0$). We present one such technique using the Barrier method.

4.3 Barrier Method

The Barrier method is used to minimize (11) under the following constraints:

$$\sum_{c_i \in \mathcal{K}(q)} \mathcal{F}(c_i) - \mathcal{F}^s(q) \leq 0 \tag{12}$$

$$\forall i: \ \mathcal{F}(c_i) > 0. \tag{13}$$

Notice that for achieving low infidelity we need not satisfy (8). Equation (8) represents the boundary condition, and hence, the equality sign can be replaced by an inequality as is done in (12). Thus, this turns out to be an inequality constrained minimization problem, which can be solved by using Interior Point methods such as Barrier method [21]. The idea is to approximate the inequality constrained minimization problem using an equality constrained minimization problem to which methods such as Newton's method [7] can be applied. This is done by making the inequality constraints implicit in the objective function using a logarithmic Indicator function (Logarithmic Barrier). Using this our new objective function is

$$\mathcal{O} = \sum_{c_i \in \mathcal{K}(q)} \left(\frac{|r_i|}{\mathcal{F}(c_i)} \right) - \frac{1}{\tau} \times \left(log\left(\mathcal{F}(q) - \sum_{c_i \in \mathcal{K}(q)} \mathcal{F}(c_i) \right) + \sum_{c_i \in \mathcal{K}(q)} log(\mathcal{F}(c_i)) \right). \tag{14}$$

The parameter τ in the above equation governs the slope of the logarithmic barrier (more on this later). Note that if any of the constraints given by (12) and (13) are violated, the value of the objective function becomes ∞. Thus, the solution (which consists of $\mathcal{F}(c_i)$ values) given by standard techniques like the Damped Newton method ensures that the constraints are not violated and the objective function is minimized. As the constraints are not violated, the infidelity is minimized.

Coming back to the parameter τ, it can be seen that with an increase in the value of τ, the indicator function approximates the ideal function in a better manner. However, if we increase the value of τ arbitrarily, it is difficult to minimize the objective function. This problem can be circumvented as follows: We start with a small value of τ and optimize (14) using the Damped Newton method. We then increase the value of τ and optimize the modified objective function, starting with the optimal value of c_i calculated during the previous iteration. This continues till a stopping criterion such as number of iterations is met.

Modifying barrier method. Notice that if the changes in Score of categories (related to a query) are in opposite directions, the use of absolute value of r_i (in (11)) can lead to a lot of unnecessary refreshes even if, overall there is no change in the top-K results. We tackle this inadequacy by increasing the value of the score-based query infidelity bound $\mathcal{F}^s$, so that the Score of all the categories can be assumed to move in one direction.

During each refresh, we compute the sum of the rate of change in the Score of those categories (minority categories) which are changing in the direction opposite to the direction of change of the Score of the query. To invert the direction of change of the minority categories, the quantity $2 \times r_i \times TTR$ is added to the score-based query infidelity bound $\mathcal{F}^s$. We have to add twice the r_i value to $\mathcal{F}^s$ to ensure that the total time required by the query to exceed the score-based query infidelity bound with the modified $\mathcal{F}^s$ and absolute r_i values (i.e., $\frac{\mathcal{F}^s + 2 \times r_i \times TTR}{\sum(|r_i|)}$) is no different than the time required with the original $\mathcal{F}^s$ and the original r_i values (i.e.,$\frac{\mathcal{F}^s}{\sum(r_i)}$). With the modified query infidelity bound, all categories can be assumed to change in the positive direction, thereby ensuring correctness of (11). The TTR that is used above is the one that we calculate using the computed infidelity limit for the category. Thus, to find the exact value of $\mathcal{F}(c_i)$ to be used in the objective function, we need to know the future TTRs. But we need the objective function to compute the future TTRs. Hence, this is a chicken and egg problem. The solution lies in the fact that our technique for minimizing (14) involves solution to a sequence of problems (one for each value of τ). The algorithm (modified Barrier method) that we use is given below:

- *Given*. Objective function $\mathcal{O}$ (14), $\nu > 1$, r_i, $\mathcal{F}(c_i)'$ (initial guess for $\mathcal{F}(q)$), τ
- *Repeat*
  - *Centering Step*. Use Damped Newton method to find the optimal value of $\mathcal{O}$ starting at $\mathcal{F}(c_i)'$.
  - *Update*. $\mathcal{F}(c_i)' = \mathcal{F}(c_i)^*$ (optimal value computed in centering step)
  - *Stopping criterion*. Quit if solution sufficiently accurate.
  - *Increase* $\mathcal{F}$. Using $\mathcal{F}(c_i)'$ in (10), find minimum TTR_{min}. Set $\mathcal{T} = \mathcal{T}' + 2 \times TTR_{min} \times r_i$ in the objective function ($\mathcal{T}' =$ original $\mathcal{T}$).
  - *Increase* τ. Set $\tau = \nu \times \tau$.

In this algorithm during each iteration, we slowly increase the value of τ using the parameter ν thereby increasing the accuracy of the logarithmic indicator function. During our experimental evaluations, we noticed that this algorithm converges very rapidly and it takes a few milliseconds to find the optimal value of the infidelity limit for a category. In summary, the constraint optimization-based algorithm used by *Caicos* for finding the infidelity limit for a category involves the following steps:

- It uses the formula given in (10) to compute the TTR for each data item.
- The infidelity limit for a category is computed using the modified Barrier method. This formulation ensures that the $\mathcal{F}(c_i)$ values do not violate (12) and (13). At the same time, it also ensures that (11) is minimized. Hence, the number of data item-category pair refreshes are minimized without affecting infidelity.

Once the $\mathcal{F}(c)$ values have been computed the next task is to find the category infidelity, which is done by the *Category Infidelity Estimator*.

5 Category Infidelity Estimator

The infidelity limit for a category is computed using an estimated rate of change in the Score of a category. This estimation is done by the *Category Infidelity Estimator*, which is outlined in this section. One can build many types of estimators; the only requirement being that they should not be costly to execute. The *Caicos* system is independent of the actual estimator used. However, for the sake of completeness, we now propose one such estimator.

To find the rate of change in the Score value, we first need to estimate the Score of a category. Hence, we first address the problem of estimating the Score value and then present a technique to find its rate of change.

5.1 Estimating Score of a Category

One way to estimate the Score of a category c for a keyword t (i.e., $\text{Score}^{est}(c,t)$) is by predicting the value of $p_c(d)$ for all the unrefreshed data items in the current window. In other words, whenever a data item d is added, we need a mechanism to predict the set of categories $C^{est}(d)$ whose predicate will evaluate to true for this data item, i.e., $C^{est}(d) = \{c \in \mathcal{C} : p_c(d) = 1\}$. We present next, a mechanism which helps us perform this prediction.

The problem with evaluating the predicate $p_c(\cdot)$ of a category c is that it is an expensive operation. One option to reduce this time is to build a (time efficient) model, which would take as input a data item d and will directly predict the set $C^{est}(d)$. Such a model can be built using the Naive Bayes technique [8]. This model will take as input the set of keywords that appear in the data item along with their frequencies and will output the set of categories to which the data item belongs. The keyword frequencies can be easily computed while doing the scan of the data item (which is done only once per data item). The accuracy of this model is likely to be low, which is inevitable due to the low processing time requirement. In spite of this, our experimental results show that *Caicos* is able to provide very low infidelity. This positive side effect can be attributed to the following two factors: 1) The infidelity limit for a category is computed in a pessimistic manner as we use the lower bound on the score-based query infidelity bound for its computation. This helps us to circumvent the error in prediction of the naive bayes technique. 2) Even if a data item d is incorrectly predicted as not belonging to category c, there will be others that might get correctly predicted as belonging to c. These correctly predicted data items help us to correct the impact of the error in prediction.

It is imperative that the time taken by our model to predict the set of categories is less than the inter data item arrival time. If this is not true, then the prediction process will start falling behind the data arrival rate. Hence, *Caicos* computes a *prediction value*, which helps it to dynamically adjust the time taken by the prediction process based on the data arrival rate. For the details of this strategy please refer to [5].

5.2 Estimating Rate of Change

Once the estimated score of category is known, the next task is to find the rate of change in the Score value. To compute this value, we have extended the core idea of Asset Pricing Model for Stocks [20]. It tries to model the behavior of a dynamically changing Score value by decomposing its changes into drift component, which is the expected change in its value (based on the history of its behavior) and a diffusion component (changes caused by processes outside the system). Our technique is based on the Black Scholes Differential Equation [20] and is given by

$$r = \mu \times dT + \sigma \times dX, \tag{15}$$

where r is the estimated rate of change in the Score value during time dT, μ is the mean of change in the Score value per unit time, σ is the volatility in the Score value, and dX is the measure of external factors. We fix dT as 1 so as to calculate the estimated change in Score value over the next time step. Every time the category is refreshed, we compute its rate of change and use an exponentially smoothed value to find μ. The value of σ is calculated using the standard formula for volatility [20]. The next task is to estimate the value of dX:

$$dX = \frac{L \times (r' - \mu_{lt})}{\sigma} + (1 - L) \times dX_{lt}. \quad (16)$$

In this equation, r' is the actual rate of change in the Score value because the last time the category was refreshed. Thus, we find the value of $\frac{r'-\mu_{lt}}{\sigma}$ (μ_{lt} is the value of μ during last category refresh), which (as per (15)) is the actual value of dX operational during the time between the last and current category refresh. Hence, this gives us a measure of the actual unexpected changes that occurred during the last two refreshes. We use an exponential smoothed value of this parameter (using smoothing constant L) to come up with an estimate of the external forces likely to be active in the future. If we now look back at (15), the first term $\mu \times dT$ helps us to estimate the drift and the second term $\sigma \times dX$ helps us in estimating the diffusion.

The above approach, however, has a potential flaw as is outlined by the following example. Consider a case where a category c_1 is the top rated category for a continuous query q and let c_k be the category at position k. Let $\mathcal{F}^s$ be the score-based query infidelity bound. In such a scenario, let the Score of both c_1 and c_k change at a constant rate of say x units per time step. Notice that a change of x in the value of c_1 is less likely to lead to a change in the top-K results (and hence infidelity) as compared to a change of x in the Score of c_k. However, if we compute the rate of change in Score value using (15), then both c_1 and c_2 would have the same rate of change and (if they have same infidelity limit) would be refreshed at the same time. As noted above, the refresh of category c_1 would be unnecessary and would not impact the infidelity. Hence, *Caicos* scales down the rate of change of those categories which are far from the kth category. For details of this strategy, please refer to [5].

Once the rate of change has been computed, the next task is to decide the set of data-item category pairs to refresh; a task done by the *Refresh Engine*.

6 Refresh Engine

The *Refresh Engine* is responsible for finding the set of data item-category pairs to be refreshed and refreshing them. We present the details of this component in the next four sections.

Caicos divides time into chunks called as epochs. The size of each epoch is decided by the *Epoch Controller*, which is explained in Section 6.4. To find the best set of data item-category pairs that can be refreshed in a given epoch, *Caicos* needs to estimate the benefit and cost of refreshing a data item-category pair—a task done by the *Cost-Benefit Estimator* (outlined in Section 6.1). Once the benefit and cost have been estimated, the *Decision Maker* uses a dynamic programming-based algorithm to find the best set of data item-category pairs (details in Section 6.2). The next challenge is to come up with an optimal schedule such that the selected data item-category pairs can be refreshed on multiple processors. This is done by the *Refresh Scheduler*, which is outlined in Section 6.3. We start by presenting the benefit and cost estimation in the next section.

6.1 Cost-Benefit Estimator

To find the best data item-category pairs to be refreshed, *Caicos* first finds the benefit and cost of refreshing a data item-category pair. The benefit of a data item d for a category c (i.e., *Benefit(c,d)*) depends on two main *factors*, namely:

1. The benefit of the category c based on the estimated $\mathrm{Score}(c, q)$ value with respect to all the continuous queries $q \in \mathcal{Q}$.
2. The benefit of the data item d based on its position with respect to the end of the window and the expiry time of the continuous queries.

Factor 1. As mentioned earlier, the *Category Infidelity Estimator* finds an estimate of the Score value for each category with respect to the continuous queries $\mathcal{Q}$ registered in the system. Hence, we can find an estimate of the reduction in the infidelity of a category due to the refresh of a data item-category pair. As our goal is to reduce the infidelity in the category, the benefit of a category (with respect to a query q) based on the estimated Score value is nothing but the reduction in infidelity of that category for q. Let $\mathrm{Score}^d(c, q)$ represent the Score of the category for the continuous query q after the category has been refreshed using the data item d. Then, the benefit of refreshing a data item-category pair due to the estimated score of the category can be represented as

$$W_{\oplus}(c, q) = \frac{(|\mathrm{Score}^d(c, q) - \mathrm{Score}(c, q)|)}{\mathcal{F}(q)}. \quad (17)$$

Factor 2. The second factor that impacts the *Benefit(c,d)* value is the position of the data item with respect to the end of the current window or the expiry time of the continuous queries. Let the benefit due to the position of the data item d, for a continuous query q, be represented by $W_{pos}(d, q)$. Refreshing a data item (d_s), which has recently been added to the window, will have a more lasting effect on the infidelity of a continuous query q as compared to refreshing a data item $d_{s-\mathcal{W}(q)+1}$, which is going to be removed from the window. Hence, the value of $W_{pos}(d_s, q)$ should be larger than that of $W_{pos}(d_{s-\mathcal{W}(q)+1}, q)$. Further, a data item can provide a benefit to a continuous query, either till the time at which the continuous query expires $F(q)$ or till the time it moves outside the window. Hence, the benefit of the data item d_i (added at time-step "i") based on its position is given by

$$W_{pos}(d_i, q) = \begin{cases} \dfrac{i - (s^* - \mathcal{W}(q) + 1) + 1}{\mathcal{W}(q)} & if\ F(q) > i + \mathcal{W}(q) - 1 \\ \dfrac{i + \mathcal{W}(q) - s^* - 1}{\mathcal{W}(q)} & if\ s^* \le F(q) \le i + \mathcal{W}(q) - 1, \end{cases}$$

where s^* in the above formula represents the current time step. Notice that $F(q) > s^*$; otherwise, q would have been removed from the system. The first of the two cases in the above formula takes into account the time till the data item $d_i$moves outside the window, whereas the second case factors in the time till the expiry of the continuous query.

Thus, W_{pos} ensures that the newly added data items have a higher benefit, which in turn increases their chances of getting refreshed. This helps *Caicos* to avoid the violation of the infidelity bound even if there is a sudden change in the characteristics of the data stream—a fact validated by our experimental results (details in Section 7).

Finally, if a category c has already been refreshed using the data item d, then refreshing it again will provide no additional benefit. Hence, let $W_{his}(c,d) = 0$ if c has already been refreshed using d; else, $W_{his}(c,d) = 1$. Thus, the benefit of a category c with respect to a data item d can be computed as

$$Benefit(c,d) = \sum_{q \in Q} [(W_{\oplus}(c,q) + W_{pos}(d,q)) \times W_{his}(c,d)]. \quad (18)$$

The above formula computes the *Benefit(c,d)* value as a sum of $W_{\oplus}(c,q)$ and $W_{pos}(d,q)$. This ensures that we give equal weight to both the factors (as opposed to using a product which would have given more weight to the factor which is more restrictive amongst the two).

In addition to the benefit of refreshing a data item-category pair, the *Decision Maker* also requires an estimate of the cost of refreshing a data item-category pair ($Cost(c,d)$). This is nothing but the time taken to evaluate the Boolean predicate ($p_c(d_i)$) associated with a category, on a data item. To compute this value, we maintain the average time taken per unit data item size by the predicate, in the past. Whenever a new data item arrives, we use this value and the size of the data item to compute $Cost(c,d)$. Once the benefit and cost have been computed, this data are fed to the *Decision Maker*, which finds the best data item-category pairs to be refreshed.

6.2 Decision Maker

Caicos divides time into chunks called epochs. During each epoch, it refreshes a set of data item-category pairs in parallel on multiple processors. Hence, if the epoch duration is 2 seconds and the refresh process is run simultaneously on three processors, then the total work done in an epoch is equivalent to the work done in $2 \times 3 = 6$ seconds on a single processor. This total work done is denoted by R and represents the time taken for refreshing the data item-category pairs selected by the *Decision Maker* on a *single processor*. Another quantity associated with an epoch is the epoch duration which is the time taken to finish the refresh task in parallel on *multiple processors* (2 seconds in the earlier example). Hence, R is typically greater than the epoch duration.

The value of R is computed adaptively by the *Epoch Controller* at the beginning of each epoch (details in Section 6.4). Thus, given the value of R for the next epoch, our goal is to identify the set of data item-category pairs such that the sum of their costs is at most R and the sum of their profits is maximized.

Organizing data items into chunks. As mentioned earlier, the *Decision Maker* only refreshes those categories whose estimated infidelity has exceeded their infidelity limit. This leads us to make two key observations:

- Our ultimate goal is to ensure that the query infidelity bound is not violated. Hence, when we select a category to be refreshed, we need to ensure that we refresh it using sufficient number of data items such that the estimated infidelity of a category becomes smaller than its infidelity limit.
- If after a refresh, the estimated infidelity becomes smaller than its limit, then as a whole the query infidelity bound will not get violated. Hence, refreshing other categories that are in the top-K of the same category would not provide us any additional reduction in infidelity.

The *Decision Maker* makes use of the above two key observations to find the set of data item-category pairs (called as a chunk) to be refreshed. The chunks are formed such that refreshing a single chunk for a continuous query ensures that the estimated infidelity of the query is within its query infidelity bound. For the details of the procedure adopted to form the chunks, please refer to [5].

Dynamic programming algorithm: The *Decision Maker* finds the best set of data item-category pairs to be refreshed. The input to the algorithm is a set of chunks each with a benefit and a cost. The chunks are also organized into subsets such that if we select any object of the subset, then the benefit of the rest of the chunks in the subset becomes zero. Further, we cannot split the chunk into smaller entities. Given this input, the algorithm needs to find the set of chunks such that their cost is less than the epoch size and their benefit is maximized. The *Decision Maker* uses a dynamic programming-based algorithm for achieving this. For details please refer to [5].

6.3 Refresh Scheduler

Caicos is designed to run in a multicore or a multiprocessor environment. We assume that there are P processors available to *Caicos*. Hence, we need to find an optimal way to schedule the refresh of the data item-category pairs present in the selected chunks, on the P processors, such that the refresh process finishes in the least possible time.

This problem is a special case of the single-stage scheduling problem where we have to schedule multiple jobs on multiple parallel machines, with each job having a job release time before which it cannot be scheduled. The goal of this problem is to minimize the longest completion time. In our case, the refresh of each data item-category pair maps to a job. Further each data item-category pair can be refreshed at any instant. Hence, the job release time of all the jobs is identical and is equal to zero. This scheduling problem is known to have a complexity which is exponential in the length of the input [11]. Hence, *Caicos* uses the approximate algorithms presented in [11] to come up with a schedule for refreshing the selected data item-category pairs on the P processors.

Once the scheduling strategy has been fixed, *Caicos* uses it to compute the epoch duration, which is nothing but the time required to finish the refresh process (as per the schedule).

6.4 Epoch Controller

The *Epoch Controller* is responsible for dynamically adjusting the value of R, which represents the total work to be done in the next epoch. Our goal is to bring the infidelity of all the registered queries within their query infidelity bound. Recall that the query infidelity can be brought within its bounds by refreshing any one of the chunks belonging to that query. Hence, the *Epoch Controller* finds the average chunk size $cost^{avg}$ and sets the R value to $cost^{avg} \times |\mathcal{Q}|$. This ensures that we try to maintain the query infidelity within their bounds for all the registered continuous queries. Thus in summary, the *Decision Maker* does the following:

- It finds the benefit and cost of each data item that helps it to identify the best data item-category pairs to refresh.
- It arranges the data items into chunks to ensure that post the refresh of a chunk, the query infidelity is within its bounds.
- The benefit and cost of the chunks is fed to a dynamic programming-based algorithm to find the best set of chunks to be refreshed.
- The refresh scheduler finds an optimal schedule to refresh the data item-category pairs on multiple processors.
- The *Epoch Controller* dynamically adjusts the size of the next epoch to ensure that the infidelity of all queries does not violate their respective infidelity bounds.

7 Experimental Evaluation

7.1 Experimental Setup

In our experimental setup, we consider a scenario where a user gives a keyword query (drawn from real world queries) over a dynamic data stream (again drawn from the real world) and is interested in knowing the top-K categories related to the query with an infidelity less than the infidelity bound. We describe next the scenario setup.

Data set. We used dynamic data from the site www.citeulike.org. This site is a free service for discovering references where people voluntarily post their academic publications from different fields such as medicine, engineering, and so on. The site offers a *"who-posted-what"* data set that contains the following information:

1. The article id,
2. the user-name of the person who posted the article,
3. the date and time when it was posted,
4. a set of tags that describe the article, and
5. the article text.

Our data set consisted of about 300,000 articles, which were posted after 30-May-2007.

We also considered another data set from Twitter that consisted of a million tweets generated in October 2009. The data also contained the time stamp at which the tweet occurred. This data set was obtained by downloading tweets generated in New York (by providing longitude and latitude range).

Continuous queries. We used queries from the publicly available Altavista query log [18] as our continuous queries. We analyzed the query log and found out the Score for all the queries in the log. Our continuous queries were made up of those queries which had the top $|\mathcal{Q}|$ scores with respect to our data set. A high Score signifies that the keywords in the query appear in a lot of documents in our data set. Hence, the Score of such queries will vary rapidly due to the moving window. Thus, selecting such queries allowed us to perform a rigorous stress test of our system.

Categories. Each article of *citeulike* is associated with a set of user specified tags. We created a category for each distinct tag in our data set. Thus, a category like say, "brain cancer" would have all the articles posted by different users related to brain cancer. Hence, the data set in our case can be assumed to be preclassified and consisted of around 3,000 categories. For the Twitter data set, we identified 3,000 categories (manually) relevant to the tweets.

Data arrival rate. In our experiments, we did a trace replay using the documents in our data set to model a continuously arriving stream of data items. The order of the data items in the stream was the same as the order in which the documents were posted on the web. The rate of arrival of data items was varied by changing the speed of the trace replay.

Categorization time. As the data items in our set were preclassified, we simulated the categorization process by running a dummy task for a duration equal to the categorization time. We evaluated different types of text classifiers as well as different types of SQL queries so as to find the typical time taken for evaluating p_c in different domains and used it in our experiments (as input to the *Refresh Scheduler*).

Processing power. Our experiments were run on a 2-GHz machine with 1-GB RAM. Recall that the processing power and the categorization time is input to the *Refresh Scheduler*, which uses it to find the epoch duration and decide the refresh (categorization) schedule. As we had preclassified data, we simulated the categorization process by adding a dummy task for a duration equal to the epoch size. An increase in the processing power would result in a smaller epoch size and hence a smaller dummy task.

7.2 Performance Evaluation

Caicos does not have any prior art that can be used for comparison of performance results. The only technique that has been advocated in the presence of dynamic data is the Uniform Refresh technique. However, in addition to the uniform refresh technique, we also compared the performance of *Caicos* with a technique, which we call as $Caicos^{-}$. $Caicos^{-}$ is similar to *Caicos* except that it does not make use of the theoretical underpinnings of *Caicos*. Specifically, $Caicos^{-}$ differs from *Caicos* in the following:

- Instead of using the modified Barrier method for finding the infidelity limit for the category, it uses a uniform allocation strategy. This strategy computes the $\mathcal{F}(c)$ values by dividing the $\mathcal{F}^s(q)$ among all the $K + \frac{K}{2}$ categories related to the query such that (8) is satisfied.
- Instead of using the Naive Bayes technique for finding the category infidelity, $Caicos^{-}$ uses a Desultory estimator that 1) randomly selected the categories related to a data item (as opposed to the

TABLE 2
Parameters Ranges and Nominal Value Used in the Experiments

Parameter	Range of Values Tested	Nominal Value
ArrRate	2 to 20 per sec	20
Processing power	33 to 116	50
Categorization Time for all categories with processing power = 1	15 to 30 sec	15
$\|\mathcal{Q}\|$	100-1000	500
Prediction Time for all categories	7 sec	7
K	10	10
α	0.7	0.7

technique outlined in Section 5.1) and 2) uses (15) to estimate the rate of change.

- $Caicos^-$ does not use the dynamic programming-based algorithm for finding the best set of data item-category pairs. It uses the cost-based strategy of selecting those chunks that have the highest benefit per unit cost.

In this section, we compare the performance of *Caicos*, $Caicos^-$ and uniform refresh strategy under different scenarios and range of parameter values. The nominal value of the various parameters used in the experiments is summarized in Table 2 (for the *citeulike* data set). We do not measure the processing power in terms of number of processors as the performance across two different processors can widely vary depending on processor speed, available memory, and so on. In our setup, we could categorize one data item for all categories in 15 seconds (total categorization time). This was considered as a baseline and mapped to a processing power of 1. In a production environment, this time would be lower due to the use of better machines. An increase in the processing power implies a reduction in the total categorization time and can map to an increase in the memory, cache, processor speed, number of processors, and so on.

In the experimental evaluation, we first found out the correct top-K categories for the continuous queries by increasing the processing power such that for each new data item, we could update the metadata for all the categories before the next data item arrived. The computed results were used to compute the infidelity of *Caicos*. In our experiments, the start and end time of the queries was (randomly) set such that we had at least 500 queries running at any given point in time.

7.2.1 Performance Using Citeulike Data Set

Infidelity with varying infidelity bound. Fig. 4 shows the variation in the infidelity with a change in the infidelity bound. As is expected the infidelity decreases with an increase in the infidelity bound (Loose = large infidelity bound). However, as the infidelity bound becomes tighter, the infidelity of the uniform refresh technique and $Caicos^-$ increases significantly as compared to that of *Caicos*. This corroborates the effectiveness of the selective update technique that helps *Caicos* to maintain its infidelity levels by refreshing the right data item-category pairs.

Infidelity with varying processing power. Fig. 5 shows the variation in the infidelity of *Caicos*, $Caicos^-$ and the uniform

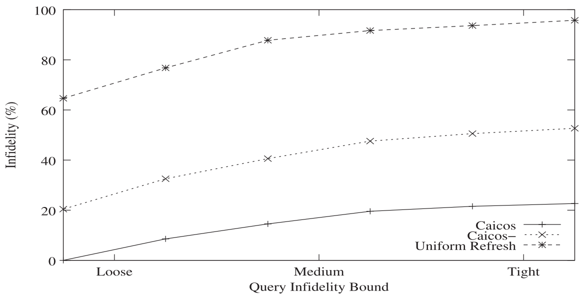


Fig. 4. Infidelity with varying Infidelity Bound.

refresh technique with increasing processing power. An increase in the processing power allows all the techniques to refresh more number of data item-category pairs, which results in an improvement in the infidelity. However, a key point which is evident from the graph is that *Caicos* can provide a user specified infidelity using significantly lower processing power as compared to the uniform refresh technique and $Caicos^-$. Specifically, *Caicos* provides an *fidelity of 99 percent using 45 percent less processing power* than the uniform refresh technique (ArrRate = 20). This demonstrates the effectiveness of the selective update technique used by *Caicos*.

Infidelity with varying data arrival rate. An increase in the data arrival rate leads to an increase in the number of unrefreshed data item-category pairs for all the three techniques. This increase leads to an increase in infidelity as shown in Fig. 5. However, the results reveal that the increase in infidelity is much more severe for the uniform refresh technique and $Caicos^-$ as compared to *Caicos*. This shows that in spite of the increase in the number of unrefreshed data items, *Caicos* is able to provide superior performance by intelligently refreshing the "important" data item-category pairs.

Infidelity with varying categorization time/Number of categories. We measured the effect of an increase in the categorization time (i.e., increase in number of categories) on infidelity. An increase in the categorization time leads to an increase in the time taken to refresh a data item-category pair, which results in lesser number of refreshes. This in turn leads to an increase in infidelity. However, the increase in infidelity was significantly smaller for *Caicos* as compared to the uniform refresh technique and $Caicos^-$. One would expect that this increase in infidelity could be countered by a proportional increase in the processing power. To validate this hypothesis, we measured the

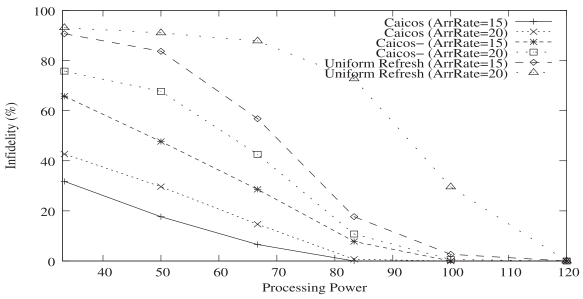


Fig. 5. Infidelity with varying data arrival rate and processing power.

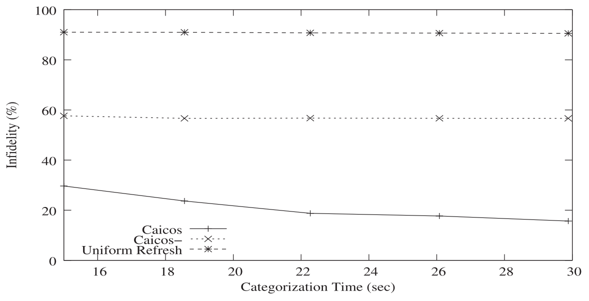


Fig. 6. Infidelity with varying categorization time.

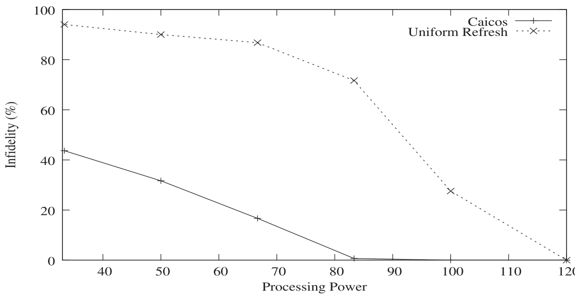


Fig. 8. Infidelity with varying processing power—Twitter data.

infidelity by varying the average categorization time and also making a proportional change in the processing power (Fig. 6). If our hypothesis was correct, then there would be no change in the infidelity as both the categorization time and the processing power are changed proportionately. In Fig. 6, as the categorization time is increased from 15 to 23 (50 percent increase) the processing power is also increased proportionately from 50 to 75. The figure shows that the infidelity of *Caicos* reduces with the increase in the categorization time, whereas the infidelity of the uniform refresh technique and $Caicos^-$ remains constant. This reduction in the infidelity can be attributed to the fact that *Caicos* makes use of the increased processing power in an optimal manner. This shows that *if the average categorization time doubles, then Caicos requires less than twice the processing power to provide the same level of infidelity*.

7.2.2 Performance Using Twitter Data Set

The key differences between the Twitter data set and the *citeulike* data set were: 1) The rate of arrival of data and 2) the time taken to categorize each data item. In the Twitter data set, the rate of arrival of data is very high (60 tweets per second). However, the classification time required for classifying each tweet is also proportionately smaller. The actual Twitter data arrival rate is much higher but our data set consists of a subset of the tweets (generated around New York) and, hence, has smaller arrival rate.

Infidelity with varying infidelity bound. Fig. 7 shows the variation in infidelity with varying infidelity bound. As with the *citeulike* data, the infidelity increases as the infidelity bound becomes tighter. But the performance of *Caicos* is significantly better than that of the uniform refresh technique.

Infidelity with varying processing power. The performance of *Caicos* with varying processing power is shown in Fig. 8 (arrival rate $= 60$). The performance of both *Caicos* and the uniform refresh technique is comparable to that observed with the *citeulike* data.

The key factor that impacts the performance is the number of unrefreshed data items. With the Twitter data set, as the categorization time is small, the time taken to refresh a single data item for all the categories is also less (as compared to the *citeulike* data). However, the rate of arrival of data items is proportionately higher, and hence, the number of unrefreshed data items (at the end of the refresh of 1 data item) is almost same in both the cases. Hence, the performance with the Twitter data is comparable to that observed in the *citeulike* data set.

The results of other evaluations were also similar to those reported in Section 7.2.1 and are, hence, omitted.

7.2.3 Additional Evaluations

Evaluation using BM25 scoring function. To test the robustness of our approach, we tested the accuracy of *Caicos* using the BM25 scoring function. Using this scoring function, the Score of a category c for a keyword query $q = \{t_1, t_2, \ldots, t_\ell\}$at time-step s is given by

$$\mathrm{Score}_s^{bm}(c,q) = \sum_{t_i \in q} idf_s(t_i) \cdot \left(\frac{(z+1)\cdot f(c,t_i)}{f(c,t_i) + z \cdot \left(1 - b + b \cdot \frac{|c|}{avgcl}\right)} \right), \tag{19}$$

where f(c, t) in the above formula represents the number of times the term t appears in the data set of c, i.e., $\sum_{d:d\in M_s(c)} f(d,t)$. $|c|$ is the size of the category c and $avgcl$ is the average size of the categories in $\mathcal{C}$. z and b are free parameters and are usually set to $z \in [1.2, 2.0]$ and $b = 0.75$. For more details please refer to [15].

Notice that the above scoring function does not account for the distance between the keywords. Hence, we alter the

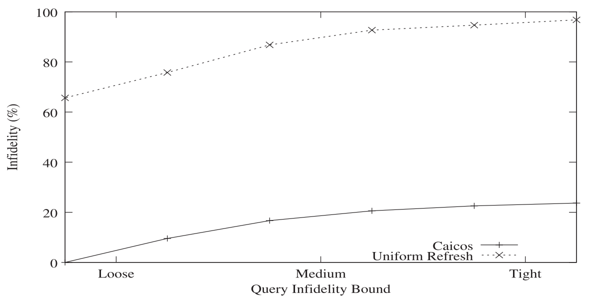


Fig. 7. Infidelity with varying Infidelity Bound—Twitter data.

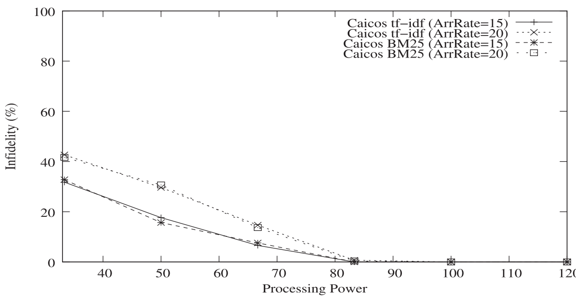


Fig. 9. Evaluation using BM25 scoring function.

above scoring function using the concept of *proximity* $(D(c, q)$: refer to (7) as follows:

$$\text{Score}_s(c, q) = \mathcal{D}(c, q) \times \text{Score}_s^{bm}. \quad (20)$$

Fig. 9 compares the performance of *Caicos* using the BM25 and $tf \cdot idf$-based scoring function. As is evident the use of BM25 scoring function does not lead to any significant change in the performance of *Caicos*. This shows that the techniques adopted by *Caicos* are general enough to be applied to different types of scoring functions.

In summary, our experimental results show the following:

- *Caicos* is able to provide an fidelity of close to 100 percent using 45 percent lower resources than the uniform refresh technique.
- *Caicos* is able to provide significantly lower infidelity as compared to that of the uniform refresh technique even in the presence of tight infidelity bounds and high data arrival rates.
- If the average time required for refreshing a data item-category pair doubles, then *Caicos* requires less than twice the processing power to maintain its infidelity.
- The performance of *Caicos* is stable across data sets.

8 Related Work

Caicos is the very first attempt at supporting Infidelity Bounded *continuous* keyword queries in a *category*-based search system over streaming data. The academic community has addressed the problem of category-based search by proposing techniques for clustering of web-search results [12]. However, there are many problems with these techniques, namely: difficulty in labeling the clusters, conflation of many dimensions, and unpredictability of the generated results [10]. Hence, *Caicos* makes use of predefined set of categories to find the top-K categories relevant to a continuous query.

One of the first works on category-based search using predefined categories appeared in [4]. However, it supported point queries. *Caicos* on the other hand addresses an altogether different class of queries, i.e., continuous infidelity bounded windowed category-based queries. The problem of efficient execution of threshold queries on structured data has been addressed in [6]. *Caicos* extends that work to the unstructured data domain and also addressed the challenges posed by categorized windowed continuous queries.

In the information retrieval domain, the problem posed by dynamic documents has been looked at from a crawler perspective, i.e., to develop better crawling techniques in the face of dynamic web content. Techniques for efficient retrieval of dynamic webpages so as to capture the maximum number of changes to the webpage have been presented in [13]. However, our algorithms focus on updating the categories once the updates to the webpages have been detected. Hence, we can make use of these techniques to detect the updates. There has also been some recent work on "discovering" emerging events from dynamic data [2], [3]. This does not make use of any keyword query and tries to identify upcoming events. Our work on the other hand, focuses on identifying categories related to a given keyword query and, hence, is orthogonal to this work.

The problem of classifying a document into multiple categories has been explored in machine learning [19], [22]. Various approaches such as ML-KNN, multilabel kernel methods, multilabel neural networks, and so on have been explored in this domain. However, these approaches can be complex and time consuming to execute and are, hence, not useful for *Caicos*. Work on keyword queries over data streams [14] might appear related to *Caicos*, but neither of these support keyword queries over categories. Further *Caicos* works on a stream of unstructured documents and, hence, significantly extends the existing state of the art in the category-based search domain.

9 Conclusions

In this paper, we presented the *Caicos* system that addresses the problem of continuous infidelity bounded windowed category-based queries over dynamic data. The system addresses those scenarios where users are not interested in finding the exact top-K answers, but are satisfied as long as the reported top-K are correct within a tolerable bound. The key challenge in answering such queries over dynamic data is that of updating the metadata in the presence of high data arrival rate. We showed that a uniform refresh technique that updates the metadata corresponding to all the categories, whenever a new data item is added, falls behind the data arrival rate due to the high cost associated with the update process. We then presented the *Caicos* system that uses a selective update technique to identify the best set of data item-category pairs to be refreshed, such that they can provide the maximum benefit to the registered continuous queries. The *Caicos* system first maps the query infidelity bound specified in terms of the top-K to the score-based query infidelity bound. It then uses a constraint optimization-based technique to find the infidelity limit for a category.

The category infidelity and the infidelity limit of a category is used by the dynamic programming-based algorithm of the *Decision Maker* to identify the best set of data item-category pairs to be refreshed. A refresh schedule for the identified pairs is found using the single-stage scheduling problem. We finally demonstrated the practicality of *Caicos* using real-world data and queries, which showed that *Caicos* is able to provide an fidelity of close to 100 percent by using 45 percent less resources than the uniform refresh technique. In summary, *Caicos* provides an efficient, robust, and scalable solution to the problem of continuous infidelity bounded windowed category-based queries over dynamic data, thereby advancing the state of the art in category-based search domain. As part of our future work, we plan to extend *Caicos* to handle in-place updates to the data items.

References

[1] E.L. Allwein, R.E. Schapire, and Y. Singer, "Reducing Multiclass to Binary: A Unifying Approach for Margin Classifiers," *J. Machine Learning Research*, vol. 1, pp. 113-141, 2000.

[2] F. Alvanaki, S. Michel, K. Ramamritham, and G. Weikum, "En Blogue - Emergent Topic Detection in Web 2.0 Stream," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* 2011.

[3] M. Agarwal, K. Ramamritham, and M. Bhide, "Real Time Discovery of Dense Clusters in Highly Dynamic Graphs," *Proc. VLDB Endowment,* vol. 5, pp. 980-991, 2012.

[4] M. Bhide, V. Chakravarthy, K. Ramamritham, and P. Roy, "Keyword Search over Dynamic Categorized Information," *Proc. Int'l Conf. Data Eng. (ICDE),* 2009.

[5] M. Bhide and K. Ramamritham, *Algorithms Used in Caicos,* www.cse.iitb.ac.in/krithi/papers/CaicosAlgo.pdf, 2013.

[6] M. Bhide, K. Ramamritham, and M. Agrawal, "Efficient Execution of Continuous Infidelity Bounded Queries over Multi-Source Streaming Data," *Proc. Int'l Conf. Distributed Computing Systems (ICDCS),* 2007.

[7] S. Boyd and L. Vandenberghe, *Convex Optimization.* Cambridge Univ. Press, 2004.

[8] D. Hand and K. Yu, "Idiot's Bayes - Not So Stupid After All?" *Int'l Statistical Rev.,* vol. 69, pp. 385-398, 2001.

[9] J.A. Hartigan, *Clustering Algorithms.* John Wiley and Sons, Inc., 1975.

[10] M.A. Hearst, "Clustering versus Faceted Categories for Information Exploration," *Comm. ACM,* vol. 49, pp. 59-61, Apr. 2006.

[11] E. Lawler, J. Lenstra, A. Rinnooy Kan, and D. Shmoys, "Sequencing and Scheduling: Algorithms and Complexity," *Handbooks in Operations Research and Management Science,* Elsevier, 1989.

[12] W. Liu, Y. Zhou, and F. Ren, "Text Clustering Based on the User Search Intention," *Physics Procedia,* vol. 22, pp. 339-346, 2011.

[13] C. Olston and M. Najork, "Web Crawling," *J. Foundations and Trends in Information Retrieval,* vol. 4, pp. 175-246, 2010.

[14] L. Qin, J. Xu Yu, L. Chang, and Y. Tao, "Scalable Keyword Search on Large Data Streams," *Proc. IEEE 25th Int'l Conf. Data Eng. (ICDE),* 2009.

[15] S.E. Robertson, S. Walker, S. Jones, and M. Gatford, "Okapi at TREC-3," *Proc. Third Text Retrieval Conf.,* 1994.

[16] G. Salton and C. Yang, "On the Specification of Term Values in Automatic Indexing," *J. Documentation,* vol. 29, pp. 351-372, 1973.

[17] D. Sifry, *State of the Blogosphere,* http://technorati.com/weblog/2006/04/96.html, 2013.

[18] C. Silverstein, M. Henzinger, and H. Marais, "Analysis of a Very Large Altavista Query Log," Technical Report 1998014, Compaq Systems Research Center, 1998.

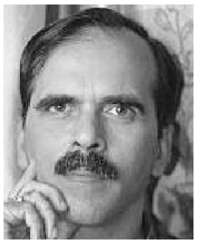
[19] G. Tsoumakas, I. Katakis, and I. Vlahavas, "Mining Multi-Label Data," *Data Mining and Knowledge Discovery Handbook,* Springer, 2010.

[20] P. Wilmott and S. Howinson, and J. Dewyne, *The Mathematics of Financial Derivatives.* Cambridge Univ. Press, 1995.

[21] M. Wright, *Interior Methods for Constrained Optimization.* Cambridge Univ. Press, 1992.

[22] M. Zang and Z. Zhou, "ML-KNN: A Lazy Learning Approach to Multi-Label Learning," *J. Pattern Recognition,* vol. 40, pp. 2038-2048, 2007.

Manish Bhide received the MTech degree in computer science from IIT Bombay in 2002, and is currently working toward the PhD degree. His research interests lie in the area of dynamic data dissemination and information integration. He is at IBM as an architect for the Information Server product.



Krithi Ramamritham is a Vijay and Sita Vashee chair professor in the Computer Science Department at IIT Bombay, India. His research explores timeliness and consistency issues in computer systems, in particular, databases, real-time systems, and distributed applications. His recent work addresses these issues in the context of dynamic data in sensor networks, embedded systems, mobile environments, and the web. He is a fellow of the IEEE, ACM, Indian National Academy of Engineering, and Indian Academy of Sciences.