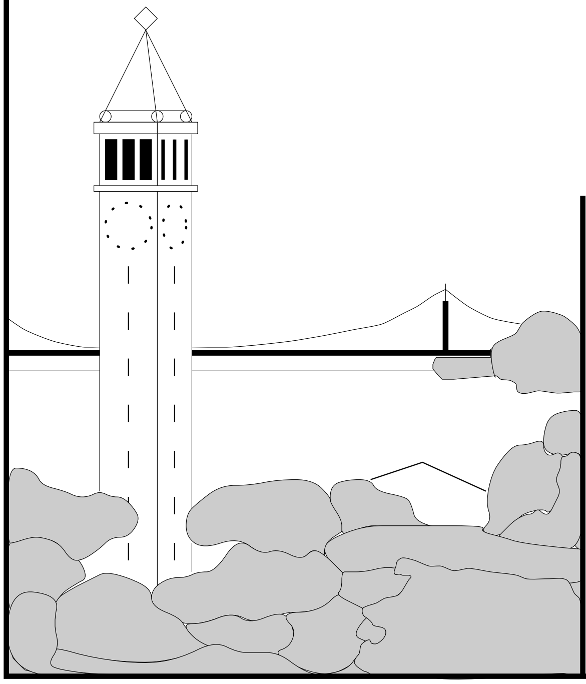
Smartseer: Continuous Queries over Citeseer

Jayanthkumar Kannan, Beverly Yang, Scott Shenker,
Puneet Sharma, Sujata Banerjee, Sujoy Basu, Sung Ju Lee



Report No. UCB/CSD-05-1371

January 2005

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Smartseer: Continuous Queries over Citeseer

Jayanthkumar Kannan, Beverly Yang, Scott Shenker,
Puneet Sharma, Sujata Banerjee, Sujoy Basu, Sung Ju Lee

January 2005

1 Introduction

In the field of Computer Science today, a research contribution is only "timestamped" if and when it is selected for publication in a conference or journal. Not only does the process of submission and publication have a long time cycle, but it is also highly unpredictable: a small number of over-worked (or lazy) program committee members shoulder the responsibility of determining which papers will have a lasting impact on the field, and which will not, in the time span of just a few hours. In contrast, in other fields such as Physics, a research contribution is recognized as soon as its preprint is made available, and the impact of the contribution is determined over a long period of time by the number of citations it receives. Thus, the process of publicizing and evaluating research contributions is much less error-prone and possibly faster as well.

To remedy the situation in Computer Science, we envision a publicly available *preprint library* in which users can submit technical reports to timestamp their contributions, and perform searches over preprint content. We name this system "SmartSeer," after the well-known CiteSeer repository of technical documents. The objective of SmartSeer has guided us in laying down two important requirements: support for rich *continuous* queries and the ability to use *opportunistic infrastructure*.

One key enabling mechanism required for such a preprint library is a mechanism for users to be notified of new documents that fall in their area of interest and expertise. For this purpose, Smartseer offers support for rich continuous queries over documents, in which users may register queries and receive notifications when relevant documents are inserted into the system. Today, researchers are alerted to new publications via conferences and journals; in an environment focused on preprints, researches must rely on these continuous queries to discover new, relevant work. Naturally, real-time (or *immediate*) queries are also supported.

Based on the current experience with Citeseer and the scale of a preprint library across all scientific fields, it is clear that Smartseer has to be distributed for scalability and fault tolerance reasons. In fact, Citeseer supports continous query functionality, but this functionality has been disabled (as has been crawling) due to excessive load. Though distributed, ideally SmartSeer could be under centralized control, with all machines under a single management at a single site with plenty of bandwidth between them. Such a configuration would allow a web service-like highly optimized design. Unfortunately, no such centralized resources are likely to be available for Smartseer. Instead, encouraged by the Planetlab model, we envision that various organizations would donate individual machines, housed at their respective sites, to the SmartSeer effort. SmartSeer must "make do" with this loosely coupled, unreliable, distributed machines – in other words, creating what we call an "opportunistic infrastructure" over whatever resources are available.

Existing work for answering continous queries in a distributed fashion are either based on tighly coupled systems (like Telegraph) or allow only simple keyword-based queries (Scribe etc). Smartseer uses a DHT-based design for continous queries, building off similar approaches for answering immediate queries (MIT paper). It also supports more expressive queries by allowing simple subqueries since it turns out that some queries of interest to users require such expressiveness. Other desirable features for SmartSeer include mechanisms for perform distributed crawling, inexact text matching etc: we consider such issues orthogonal to our main thrust, and research topics in their own right. Smartseer has been deployed and tested over Planetlab.

Although Smartseer has been designed with a specific application in mind, it also offers design insights into executing continous queries over DHTs. A simple model shows that when implemented over a DHT, continuous queries have intrinsically different characteristiscs and scaling properties compared to immediate queries. Morever, latency expectations for continuous queries are typically less stringent compared to immediate queries, and our system allows for optimizations that exploit this characteristic. We also believe Smartseer also offers design guidelines for larger scale system such as web alerts, news alerts, etc.

Paper Outline. In this paper, we present the basic architecture of the SmartSeer system (Section 2), along with an analysis of the new technical challenges faced by our system (Section 3). We then validate our design decisions using simulation analysis of real-life workloads (Section 4). Finally, we report on our initial experience in deploying SmartSeer as a publicly available service (Section 5). Our goal for this paper is not only to describe the technical aspects of SmartSeer, but also to raise awareness of and participation in the current SmartSeer collaboration.

2 Basic Architecture

Smartseer supports keyword search over text documents and metadata using standard information retrieval techniques. Queries consist of *search terms*, which are keywords that appear in an optionally specified context (e.g., “title:smartseer” specifies the keyword “smartseer” appearing in the title of the document). Boolean conjunctive and disjunctive queries are supported, as well as simple subqueries for more complex constraints. For example, we may ask for documents by all co-authors of Jane by the query “author:(author:Jane).” Because support for OR queries and especially subqueries is relatively complex, we focus primarily only on the support of simple conjunctive queries.

Our basic design is based on the simple observation (pointed out in Telegraph) that executing continuous queries can be thought of running immediate queries with the role of documents and queries reversed. Queries are stored in the system, and on the arrival of a new document, queries that match the document are notified of the same.

2.1 DHT-based Architecture

There are three primary design choices for a loosely coupled distributed system for continous queries: mirroring, partition-by-ID, and partition-by-keyword. In the query mirroring approach, all documents and continous queries are stored on all nodes, and a new document or immediate query is sent to a randomly chosen mirror. The partition-by-query approach partitions the continous queries or documents among the nodes, and a new document or immediate query is sent to all nodes. The partition-by-keyword method builds a distributed index of the keywords in the continuous queries or documents using a DHT. As it turns out, support for certain kinds of continuous queries requires the ability to support immediate queries as well. Thus, the design choice for Smartseer is dictated by the constraints imposed by both kinds of queries.

Firstly, we rule out the mirroring option: though it might work for smaller data sets, it does not scale to Smartseer’s requirements. For example, the current CiteSeer database has over 700GB worth of document data. Not only would we expect a preprint library to exceed this size by orders of magnitude (especially if topics outside of Computer Science are included), but in an opportunistic infrastructure we need to make use of the available resources, which will likely not be powerful servers with terabytes of storage. Query mirroring also may not be suitable for Smartseer if a huge volume of continous queries are registered.

Secondly, notice that in the partition-by-keyword approach, during the insertion of a new document, only nodes that store queries containing keywords belong to the document need to be contacted. In contrast, in the partition-by-ID approach, irrespective of the number of nodes in the system, all of them will have to be notified of the arrival of the new document. We expect that queries will mostly use words that occur in the meta-data of the document and the abstract of the document.

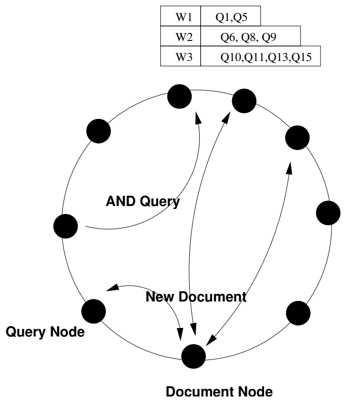


Figure 1: Basic Architecture

For this expected common case, the partition-by-keyword approach provides better scaling bandwidth since the number of nodes that needs to be contacted will be typically lower (assuming that the infrastructure consists of order of hundreds of machines).

For these reason, Smartseer is based on a DHT design to achieve partition-by-keyword. Basing the design upon DHT allows us to build a self-organizing, self-healing system over opportunistic infrastructure. As we will describe later, SmartSeer employs standard P2P techniques in its architecture, and extends these techniques to cover the functionality we wish to support.

2.2 Continuous Queries

Our architecture for supporting continuous queries is a simple extension of the conventional architecture used for executing immediate queries over DHTs. For now, we focus on simple conjuctive queries that involve multiple terms of the form “author:name” and “text:databases” etc.

Every DHT node is responsible for maintaining a list of queries whose keys fall into the keyspace of that node. In particular, smartseer nodes participate in a distributed hash table (DHT), using the Bamboo DHT, which is based on Pastry. Distributed hash tables provide a simple *put/get* interface that allow applications to insert and retrieve objects by key over distributed, unreliable storage. Nodes in a DHT are automatically assigned a region of an identifier space for which they are responsible. Responsibility of terms is assigned by hashing a term, and assigning it to the node responsible for the space in which the hash falls.

The key of a query is defined to be the hash of the *most selective* term t_s in the query. We say that the query is *registered* on term t_s, because it is stored at the node responsible for t_s, and will be processed only if a document contains t_s. Statistics about selectivity of the different terms can be obtained in our system by querying the node responsible for storing the inverted list of *documents* that contain the term. Thus, for conjunctive queries, a query and its metadata is stored only at a single node. This is unlike document indexes, where only the ID is stored in many inverted lists, queries are *registered* on only a single term.

Now,we describe the document insertion process. This process is shown in Figure 1. When a new document is inserted, all nodes that are responsible for some term in the document are contacted. We refer to the node inserting the document as the *document insertion* node, and each of the nodes contacted as the *query* nodes. When a document is inserted into the system, one node serves as the document insertion node, and is responsible for handling all actions associated with the insert. A document insertion node may simply be the document to which a document is initially uploaded, or it may be chosen for particular properties. The document insertion node will parse the document into *tokens*, which are then transformed into terms via stop-word filtering and stemming (filtering and stemming is performed for query terms as well).

Due to the fact that the entire query is stored at the query nodes, each continuous query can be processed *locally* – there is no need to ship inverted lists as with immediate queries. However, to process a query locally, a node needs to know whether the keywords in the query appear in the document. We discuss this need shortly. Second, by registering on the most selective term, we seek to maximize load-balancing, and minimize wasted processing. If queries were registered on common terms, then the node(s) responsible for the most common terms would have disproportionately high load, and would process many spurious results. These spurious results would occur because the size of the inverted list for popular keywords would tend to be higher.

To enable a node to process its continuous queries on the insertion of a new document, we must somehow send sufficient information about the document over to these nodes. For example, if the query "cat dog aardvark" were stored on the node responsible for "aardvark," that node must also know whether "cat" and "dog" appear in the document, before it can determine whether the query is satisfied by the document. As a baseline technique, we may choose to send the entire document to each node in N, who can then use the keyword information present in the document to evaluate the continuous queries. Clearly, sending the entire document may not always be a wise choice with respect to efficiency. In Section 3, we discuss different approaches to processing continuous queries.

2.2.1 *OR queries and Complex queries.*

OR queries are registered on every term, rather than the least selective. When a document is inserted into SmartSeer, if it contains any terms in the OR query, then the query will be processed. If the document contains more than one term in the query, the query may be processed multiple times.

Continuous complex queries are slightly more complicated. Firstly, on query insertion, the subquery is run as an *immediate query*, and the subquery is said to have been *translated.* The translated sub-query is an OR-query consisting of all the results returned by the subquery. Once all sub-queries have been translated, the query is now registered at all terms or one of them (depending on whether it is a OR query or an AND query). In addition, the original subquery itself is registered, so that so that new results to the subquery are obtained. Such new results are sent to the node(s) responsible for storing parent query.

Consider our example from the previous section: YEAR:2004 AUTHOR:(networks AUTHOR=Bob). The original subquery is "networks AUTHOR=Bob" and the translated subquery is "AUTHOR:AuthorX OR AUTHOR:AuthorY OR AUTHOR:AuthorZ." Say a new document is inserted in 2004, in which one of the authors is AuthorZ. Assume that the query is registered on all terms in the translated subquery. This document will cause the translated subquery to be processed, which, when found to have been satisfied, will in turn trigger the original query for processing. If the translated subquery were not registered, then for every document inserted, we would have to re-evaluate the entire subquery and query expression as described in the previous section. The translated subquery is essentially a "materialized view" of the results of the subquery.

However, registering the translated subquery is not sufficient. Say Carol has never written a paper with Bob before on networks, until recently. When their document is inserted, Carol now satisfies the subquery, and suddenly all of Carol's documents from 2004 now satisfy the query. From this example, we can see how with subqueries, old/pre-existing documents can be returned as continuous query results when a different document is inserted.

As noted in previous literature, negated predicates are problematic for materialized views, for which continuous queries are a special case. In particular, a negated subquery could require that certain results for a continuous query be "recalled." As there is no clean way to handle this problem, SmartSeer does not allow negated terms in continuous complex queries.

2.3 Immediate Queries

We employ the standard approach to supporting immediate keyword queries over DHTs. We use an *inverted index*, where for every term in the corpus, there exists an *inverted list* of *postings* specifying the documents in which this term appears, and the number of occurrences of the term within each document.

To distributed this inverted index, SmartSeer employs the "partition by keyword" approach (cite MIT paper) where nodes are responsible for a subset of terms appearing in the corpus, and each node manages the inverted lists corresponding to those terms. The *document insertion* node described in the previous section is also responsible for updating document indices.

When a user submits a query to the system, again, one node serves as the *query insertion* node. For each term in the query, the query insertion node retrieves the corresponding inverted list by hashing the term and again using the DHT to find the node responsible for the inverted list. These lists are then merged to answer the query. Note that optimizations such as those described in (cite MIT paper) can be employed to decrease bandwidth consumption in this step.

In our system, the document index update is piggybacked on the query notification message. Because a query q is stored on a node responsible for at least one term in q, q is necessarily co-located with the inverted list of that term. As a result, when a new document is inserted into the system, the nodes that manage the inverted lists relevant to that document are the same set of nodes that manage all queries with at least one keyword appearing in the document. Therefore, to update document indices on the insertion of a new document, we do not need to contact any nodes outside of those query nodes notified.

2.3.1 OR queries and Complex queries.

OR queries are handled in a fashion similar to AND queries, except that a document is returned as a result if it is found to appear in *any* inverted list, rather than all.

When a query contains a subquery term, we first execute the subquery to retrieve a list of documents D that satisfy the subquery. From these documents, fields of the appropriate type are extracted. An OR query is then created that represents the materialized, or *translated*, version of the original subquery. This translated version is then executed to retrieve a list of documents D' that satisfy the term. D' acts as an inverted list for this term, and may then be used in processing the original query.

For example, say a user submits the following query: "YEAR:2004 AUTHOR:(networks AUTHOR:Bob)" – in other words, all papers written in 2004, for which at least one author has co-written a paper with Bob containing the keyword 'networks.' Say the only authors that wrote a paper containing 'networks' with Bob were AuthorX, AuthorY and AuthorZ. The translated version of this subquery is thus AUTHOR:AuthorX OR AUTHOR:AuthorY OR AUTHOR:AuthorZ. Clearly, given the existing data (at the time the query is first submitted), this translated subquery is equivalent to the original subquery.

To execute the above query, SmartSeer will first execute the subquery, to get all documents D written by Bob containing the term 'networks.' Given these documents, SmartSeer will then extract the authors from these documents (AuthorX, AuthorY, and AuthorZ), and create the translated OR query as described above. SmartSeer will then execute the translated subquery, to retrieve all documents D' by one or more of these three authors. SmartSeer will also retrieve the inverted list I for "YEAR:2004." By taking the intersection of I and D', the final result is produced.

2.4 Expressiveness

It is clear from the discussion so far that continous queries and immediate queries are executed in our system based on a rendezvous mechanism: documents and queries that share a keyword both update the same DHT node, and a match is discovered. This rendevzous mechanism has the following consequences regarding the expressiveness of the queries.

Range queries are especially hard to implement in this mechanism, since all nodes that store keys in the range have to be notified. For immediate queries, contacting all nodes in this range in unavoidable. If required, range queries can be implemented by some form of multicasting (which would require high bandwidth). Other limits on the expressiveness of queries in Smartseer (as compareed to a query language like SQL) are that conditions that involve comparison of attributes in the relation cannot be supported efficiently. The same holds for similarity search operations similar to supported by Citeseer today (these however can be supported better by a pSearch-like architecture at the expense of accuracy).

In general, our stance on these limits of expressiveness is that solutions that do not have such limits are orthogonal to the objective of Smartseer, and can be plugged into our system if required. Currently, Smartseer has two options for dealing with such expensive queries. The first option is that if there are other simpler predicates in the query that are already directly supported in our system, then we decompose the query into an "easy" part and "hard" part: the "easy" part is registered in the DHT, and the "hard" part is verified on a hit. For example, for conjunctive continuous queries that have a subquery as one of its terms, if there are other simple terms in the query, then registering the query on one of these other terms might be a more efficient solution. We expect this option to be sufficient for the workload that Smartseer is geared towards. The second option is store such expensive queries on a smaller set of machines, to which all new documents are sent.

2.5 Sample Queries

In order to motivate the need for expressive queries in SmartSeer, we list some sample queries below. One kind of query are queries based on the document text (*e.g.,* documents containing the words "tcp" and "wireless"). We expect these to be relatively simple queries. Users of the system will also be interested in tracking citations to their papers. One way for an user to do this would be: if he wants to be notified of new citations to one of his papers, say with ID=X, then he could insert the following continous query: "X in Document.CITATIONS". If an user would like to track citations to *all* his papers, he can insert one such query for each of his papers. Or he could use subqueries and insert the following query: "X in Document.CITATIONS and X IN (AUTHOR=AUTHOR_NAME)". Thus, subqueries could be certainly of use to users of Smartseer. For another example, consider an user who is interested of keeping track of new papers written by his co-authors: the query "Document.AUTHOR =

Y and Y in AUTHOR (AUTHOR=X)" would notify the user of papers written by co-authors of X.

3 Design Issues in Continuous Queries

While much work has focused on supporting immediate keyword queries in a peer-to-peer setting (cite lots of papers), we do not know of any prior work on continuous keyword queries. Here, we discuss the challenges in supporting continuous keyword queries, and a number of techniques we use to address these challenges. Later in Section 4, we evaluate our techniques over realistic workloads, and show that proper choice of technique can greatly improve the efficiency of handling continuous queries.

The baseline method for executing continous queries parallels the one typically used in immediate queries: the document insertion node fetches the inverted list of queries stored under all terms in the document, and then triggers further action. We refer to this baseline method as the "fetch queries" method. We first discuss the need for improving on this baseline method, before discussing further optimizations.

3.1 Scaling Properties

In this section, we analyze the bandwidth consumption in answering continous queries. Firstly, we assume that the method used in answering continous queries is similar to that used in answering immediate queries over DHTs: fetch the inverted lists of queries stored on each term in the document, and merge these lists.

A detailed analysis on the lines of the MIT paper can be used to estimate the bandwidth required in such a method. The bandwidth required to handle immediate queries (at the rate of I) over N documents over a DHT is $\alpha \times NI \times W_q W_d$ (where α is a contant depending on the distribution of terms in queries and documents, W_q and W_d are the number of terms in queries and documents respectively). As a first approximation, the bandwidth required to handle C continous queries when new documents arrive at a rate R, is $\alpha \times RC \times W_q F W_d$. If all the queries are conjunctive queries, W_q can be dropped, F is a factor representing the fraction of words in a document that are actually used in queries.

Substituting typical values (rate of new documents is set at 1000-10000 documents per day) suggests that the cost of handling new documents is 3 orders of magnitude easier than answering immediate queries. Thus the negative result in the MIT paper for immediate queries over DHTs might not hold for continous queries. Note however, that this bandwidth is still high, and optimizations that cut down on this bandwidth are useful.

However, we wish to point out that DHT systems for answering continous queries differ in one main aspect from those for answering immediate queries: the former indexes queries, while the latter indexes documents. Since queries are typically much smaller than a document, it is feasible to store the *entire* query at its terms, instead of its identifier. Thus, the inverted lists at each term stores all the queries registered at that term. With this kind of storage, when a new document is inserted, the *entire* document is sent to the nodes storing the inverted lists. Since these nodes store all terms in the query, and are informed of all terms in the document, then it can perform notification. The possibility of such strategies (which we will discuss later) leads to the following important fact. The bandwidth consumed by a DHT-based immediate query system increases without bound as the size of the document corpus increases. On the other hand, as the number of continous queries increases, the entire document could be sent over instead of sending the queries, thus the bandwidth consumed is at most the cost of sending the entire document to all the nodes in the system.

3.2 Joining Document and Query Metadata

To answer a continuous query on the arrival of a new document, the first challenge is to somehow join information regarding terms in the query and terms in the document. Two characteristics of continous queries help us improve over the "fetch queries" method: the entire query is stored in the inverted list and latency is not a stringent constraint. The primary objective of optimizing the document insertion process is to reduce bandwidth, which is the main bottleneck resource. We also wish to split the storage and communication load nearly equally across all the nodes in the system. Also, all these optimizations are geared towards conjunctive queries, since an OR query can be notified right away. Apart from the "fetch queries" option, there are three other possible options:

- **Send Document:** The document insertion node sends the entire document to the nodes containing continuous queries relating to terms in the document (the "query" nodes'). The query nodes then check for matching queries, and trigger further action.

- **Term-by-Term Dialogue:** In a *term-by-term* dialogue, the query node sends a message to the document node asking about the presence of a set of terms in the document, and the document insertion node responds with a bit vector – one bit per requested term – specifying the presence of each terms. We say that a term is *resolved* if it the query node includes it in the dialogue, and the document insertion node responds regarding its presence. On one extreme, the query node can resolve all distinct terms appearing in the queries in a single message; at the other extreme, the query node can resolve a single term at a time. Due to packet header overhead, *batching* terms is desirable.

- **Send Bloom Filter:** The document insertion node sends a bloom filter of all its terms to the query nodes. Since the bloom filter has no false negatives, the query node can discard queries that have a term corresponding to a 0 in the bloom filter. Thus, the query nodes prune down the set of queries stored in the inverted list to a potentially smaller set. At this point, the query nodes can send all these queries to the document insertion node, or initiate a term-by-term dialogue.

Note that each of these methods is more efficient than the "Send Queries" option since the query metadata (e.g., information on the user who registered the query, feedback information used to tune relevance ranking, etc.) is typically much larger than the terms comprising the query. As a result, rather than shipping an entire query over to the document insertion node, it always makes sense to instead batch all terms into a single round of a term-by-term dialogue.

These three approaches can all be optimal in different scenarios. Here, we present a model for the bandwidth costs of each approach that will show us the general scenarios in which each approach is optimal. In our workload simulations 4 we demonstrate the utility of this model in determining the best approach.

3.2.1 Modelling cost

In this section, we provide an approximate model for the bandwidth consumed. All costs below are in terms of the number of terms that must be shipped. As a result, absolute bandwidth cost would multiply the below costs by $\bar{W}$, the average length of a term (in bits).

First, the bandwidth cost SD of shipping a document is a simple constant:

$$SD = d \tag{1}$$

where d is the number of unique terms in the document.

The cost TD of a term-by-term dialogue is, in the worst case:

$$TD = q \tag{2}$$

where q is the number of unique terms across the queries. In practice, TD may be much smaller than q, due to the possibility of *query elimination*. Assuming conjunctive queries, if a term t is resolved and found not to appear in the document, then all queries containing t may be eliminated from consideration, since they can not possibly be satisfied. For any term t', if all queries in which t' appears have been eliminated, then it is unnecessary for t' to be resolved. In the extreme case, if every query contains the term t which is found not to be present, then all queries are eliminated. The cost of the dialogue is simply the size of two small messages. Thus, the actual cost of the dialogue is:

$$TD = \delta \cdot q \tag{3}$$

where δ is a fudge factor depending on which terms are actually present in the document, and the order in which terms are resolved.

There are many opportunities to optimize the term-by-term dialogue. Given general probabilities of the likelihood of terms appearing in documents, a query node may first resolve terms that are not frequent, or terms that appear in many queries, in order to maximize the expected number of queries that can be eliminated. However, due to packet header and messaging overhead, it is also important not to extend the dialogue to too many rounds. The topic of optimizing term-by-term dialogue is covered later in this section. For now, we use the worst-case cost described above.

The bandwidth cost BF of shipping a bloom filter, followed by a term-by-term dialogue for all non-filtered query terms, is:

$$BF = f(d) + \epsilon \cdot q + \gamma \cdot q \tag{4}$$

where $f(d)$ is a function determining the "optimal" bloom filter size given a document size, q is the number of unique terms present in the queries, ϵ is the error rate of the bloom filter – the percentage of false positives, and γ is the fraction of query terms that do appear in the document. Note that due to the possibility of a false positive, any term that "passes" the bloom filter must still be resolved in the term-by-term dialogue. Hence, $\epsilon \cdot q + \gamma \cdot q$ represents the number of terms that pass the filter, including both false and true positives. Note that, again, we are assuming the worst-case cost for the term-by-term dialogue in which no queries are eliminated. However, given the low false-positive rate of bloom filters, the worst case for this term-by-term dialogue is likely to be very close to the average case.

For a given false positive rate ϵ, the size of the optimal bloom filter is given (approximately) by $0.625d\log(1/\epsilon)$. So, the equation becomes:

$$BF = 0.625d\log(1/\epsilon) + \epsilon \cdot q_1 + \gamma \cdot q \tag{5}$$

where q_1 is the number of queries that have all terms matching but one. q_1 can be written in terms of the selectivity of the terms in the query.

Looking at the above equations, the basic tradeoff between the three approaches is clear. If d is large relative to q, then the dialogue is best, given that it is not a function of document size. If q is very large relative to d, then depending on the values of ϵ and γ, shipping the document or bloom filter would be best. If we assume that ϵ, the false positive rate, is small given an appropriately sized bloom filter, and that the size of the bloom filter, $f(d)$, is a slowly growing function of document size, then shipping a bloom filter is better than shipping the full document if the fraction of satisfied queries (roughly γ) is low. Otherwise, if γ is relatively high (e.g., if queries tend to have just a few common terms), then shipping the document is best. Later in Section 4, we will quantitatively compare these approaches through simulations of SmartSeer over typical document workloads, and show how the above simple model predicts well the best approach to use.

Note that in all approaches, the document insertion node must first "notify" the query node of a new document, and of the need to process continuous queries over this document. This *document notification* message may also contain the bloom filter, document, or whatever information is necessary to implement the desired join approach.

3.2.2 *Optimization of Term-by-Term Dialogue*

Unless we choose the ship the document to the query nodes, our approach to joining query and document metadata will involve a term-by-term dialogue. For this reason, here we consider how to optimize this dialogue. For simplicity, we will initially ignore the issue of packet headers, and our goal will be to minimize the number of terms resolved. By eliminating queries, as discussed earlier, we may be able to eliminate terms from consideration as well. A query node must therefore order the terms to be resolved so as to maximize the number of expected terms eliminated.

The exact solution to this problem seems hard (more specifically, it sems to involve computation exponential in the number of rounds in the dialogue). For this reason, we consider heuristics to optimize the term-by-term dialogue. First, we observe that there is a class of terms that cannot be eliminated. These are the terms that appear in any remaining query (i.e., query that has not been eliminated) such that all other terms in the query have already been resolved. In order to determine whether the query is satisfied by the document, this last remaining term must be resolved. Therefore, an optimal dialogue will always resolve these "singleton" terms first. Once a singleton term is resolved, new singleton terms may be introduced. For example, consider the queries "cat dog" and "dog cow." If "cat" is resolved and found to be present, "dog" becomes a singleton term, but "cow" does not. If "dog" is subsequently resolved and found to be present, then "cow" becomes a singleton term as well. Therefore, an optimal dialogue will repeatedly resolve singleton terms until no such terms remain – that is, all remaining queries have 2 or more unresolved terms. It is possible to develop other heuristics based on the selectivity of the terms and the number of queries containing a specific term: we leave the detailed investigation of such queries for future work.

3.3 Document Notification

In a large-scale system with thousands or even million of nodes, each term in a document will likely hash to a separate node. However, in a likely implementation of SmartSeer on the order of tens or hundreds of nodes, there will be significant overlap in the nodes to which terms hash. Therefore, it may not make sense to process continuous queries on a per-term basis, as described above. Instead, we may wish to *batch* process queries by node.

For example, if terms "cat," "dog" and "cow" all hash to the same node, then that node will process all three sets of corresponding queries at once. Consider the scenario in which documents are shipped to query nodes to process continuous queries. By recognizing that the above three terms hash to a single node, only a single document need to be shipped to the node, rather than three copies of the same document.

In the *clustered* approach to document notification, the document insertion node will first find, for each term, the query node responsible that term. The document insertion node will then send one document notification message to each unique node, along with the terms in the document that each node is responsible for. For example, if we use the term-by-term dialogue to process continuous queries, the document notification message for the example node above would contain the terms "cat," "dog," "cow," and the number of occurences of each term in the document (so that the query node can update the corresponding document inverted lists that it also stores). Note however that this clustered approach is harder and complex to code robustly due to the fact that keyspace mappings may change during the notification process. In our implementation, clustering is implemented by looking up terms in a serial fashion: looking up a term gives the node responsible for the term, as well as the keyspace that node is responsible for.

If the number of nodes in the system is very small compared to the number of unique terms in a document – e.g., 20 nodes, and 2000 unique documents – then with high probability, every node will receive a document notification message. In this case, rather than performing a lookup for every term in the document, the *broadcast* approach to document notification will simply broadcast the notification message to all other nodes in the DHT, via an efficient application-level multicast tree (cite... was it herald?).

However, broadcast cannot be used for all join approaches. In particular, it cannot be paired with a term-by-term dialogue, since a query node receiving the notification will have no ideas which terms in its keyspace are relevant to the document. Similarly, broadcast cannot be paired with bloom filters. Broadcast is only an option if we choose to ship the document to the query nodes. With the full document, the query nodes can extract which terms in its keyspace are relevant to the document, and thereby update the appropriate inverted lists and process the appropriate queries.

4 Workload Simulations

In this section we describe simulations of SmartSeer over realistic workloads, and quantify the comparison across the different approaches to supporting continuous queries described in the previous section.

4.1 Experimental Setup

Our Smartseer implementation is written in Java using the libraries exported by the Bamboo and OpenHash code. This was deployed on Planetlab, and the same code can be used for simulating a number of nodes on a single node.

We run SmartSeer over two document sets: CiteSeer (cite), and TREC (cite). CiteSeer is our target application, so it is the best-suited corpus over which to simulate SmartSeer. However, since the SmartSeer architecture and techniques may be applied to different application scenarios, we also study the performance of SmartSeer over TREC, a widely-used corpus for evaluation of information retrieval systems. The main difference between these corpora is that the TREC data consists of smaller documents (an average of about 200 unique words)

Parameter	Default
Network size	10 nodes
Document Set	CiteSeer
Query Workload	Generated
Join Approach	Ship document
Notification Type	Naive
Mean terms per query	5
Skew type	Same
Number of continuous queries	50000

Table 1: Default parameter values for simulation

compared to Citeseer where the document had on an average about 2000 unique words. For this reason, we use a bloom filter of $10K$ bits for the CiteSeer data, and a bloom filter of $1K$ bits for the TREC data. In both cases, we inserted 1000 random documents from the corpus.

We run SmartSeer over two types of query workloads: queries submitted to the MIT website, and synthetically generated queries. Queries submitted to the MIT website comprise a realistic workload for immediate queries; however, it is not clear the continuous queries will exhibit the same properties as immediate ones. Therefore, we also generate synthetic query workloads with different properties we can tune. Queries are generated in the following manner: First, we calculate the term frequency distribution over the document corpus. We then generate a query by selecting terms independently from this distribution, without replacement. The number of terms for this query is generated from a normal distribution with mean ν and standard deviation $.3\nu$. Before generating queries, we may perturb the distribution by increasing the skew (i.e., those terms that appear frequently in documents appear even more frequently in queries), maintaining the same skew, ignoring the skew (i.e., using a uniform frequency distribution), or inverting the skew (i.e., those terms that appear frequently in documents appear infrequently in queries). The first 500 documents are used to infer the term frequency distributions, and the next batch is used to measure the bandwidth consumed. Because no real continuous query workloads are available, it is difficult to project the properties of term distributions within continuous queries.

The above description query generation involves several parameters. In addition, several other simulation parameters, such as network size and join approach, must also be specified. Unless otherwise specified, parameter values are set as shown in Table 1.

4.2 Comparing the Join Approaches

First, let us compare the different join approaches under different workload scenario. Figure 2 shows the number of bytes required for each approach along the y-axis, where the number of distinct query terms (the term q from our analysis in Section 3) is varied along the x-axis. Figure 3 plots the number of bytes versus the number of results.

In these figure we see clearly the basic tradeoffs described in Section 3. When q, the number of query terms, is very low, the term dialogue has best performance. As q grows, however, the cost of the term dialogue grows linearly with q. Likewise, the cost of bloom filter follows by a dialogue grows with q, the constant $(\epsilon + \gamma)$ is so small, this approach still has good performance even when q is fairly large. In our simulations under the default parameter values, we did not reach the point where bloom filter has worse performance than shipping the document.

4.3 More Queries

In order to see how each of these methods scales with respect to the number of queries, we doubled the number of queries and the results are plotted in Figure 4. The "Send Document" method was left out of the analysis since it does not vary with this parameter. Clearly, Bloom filter performs better than the term-by-term dialogue since the constant δ is greater than $\epsilon + \gamma$.

4.4 Number of terms

The average number of terms per query was also varied to $2, 4, 6, 8$ terms. The "Send Document" method does not have any variation with this parameter. The term-by-term dialogue and bloom filter however do vary: this variation is plotted in Figure 5 for the TREC data and in Figure 6 for the Citeseer data. As can be seen, as the number of terms increases, the Bloom Filter performs better than the term-by-term dialogue. At 2 terms query however, the term-by-term dialogue performs better. This effect is mainly due to the fact that as the number of terms decreases, q_1 increases, and thus the bloom filter pays a greater penalty for a false positive.

Recall that bloom filters perform relatively poorly when γ is large – in other words, when a large fraction of queries are satisfied by the document. The probability of satisfaction is roughly inversely proportional to the number of terms in a query. the slope of the bloom filter curve increases when terms per query is small. When the number of query terms is large and queries have 2 terms on average, the cost of the bloom filter approach exceeds the document-shipping approach. Therefore, in a system in which many continuous queries are registered (implying large q) and queries are easily satisfied (implying large γ), shipping document outperforms shipping bloom filters.

4.5 Distribution of terms

Skew of terms also has an effect on the best choice of join approach. We have experimented with two other kinds of skews: the uniform distribution and the inverted skew distribution. The results for the uniform distribution are shown in Figure 7. As can be seen, since the domain of terms is relatively large, the inverted lists are usually short, and the term-by-term dialogue wins.

Results for the case when the query distribution of terms is the inverse of the document distribution of terms are shown in

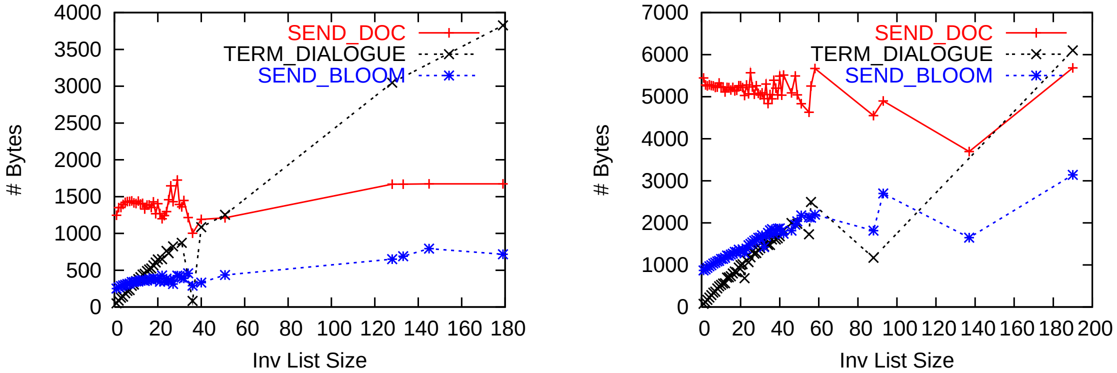


Figure 2: Bytes Vs Inverted List Size (a) TREC data (b) Citeseer

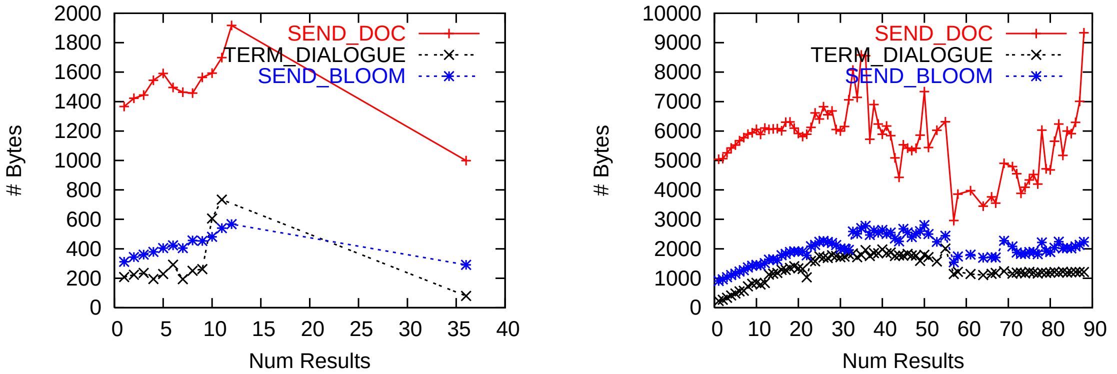


Figure 3: Bytes Vs Number of Results (a) TREC data (b) Citeseer

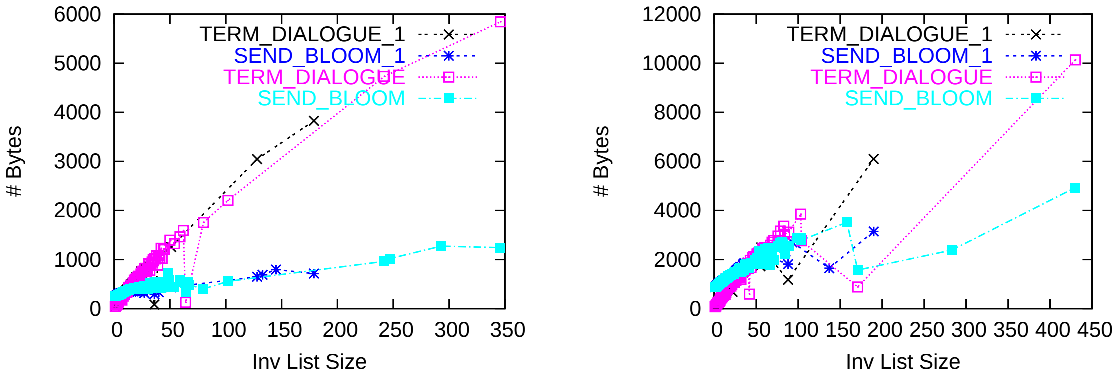


Figure 4: (More Queries) Bytes Vs Inverted List Size (a) TREC data (b) Citeseer

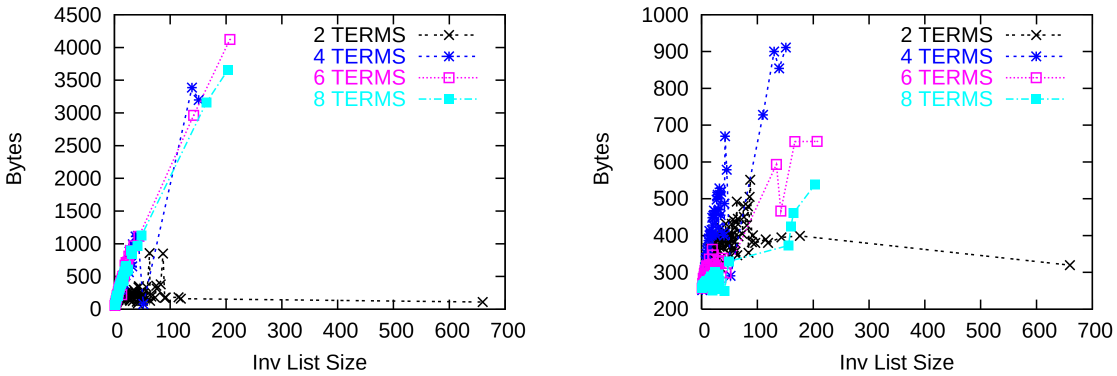


Figure 5: (Number of Terms) Bytes Vs Inverted List Size (TREC data) (a) Term-by-Term Dialogue (b) Bloom Filter

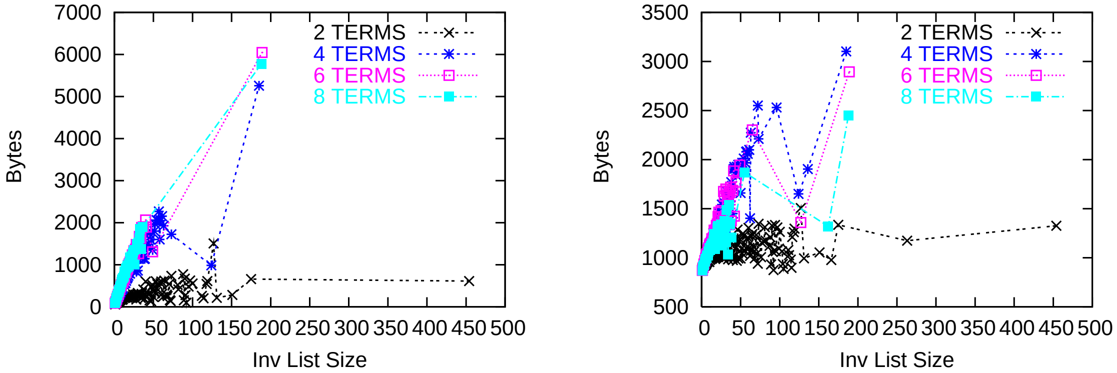


Figure 6: (Number of Terms) Bytes Vs Inverted List Size (Citeseer data) (a) Term-by-Term Dialogue (b) Bloom Filter

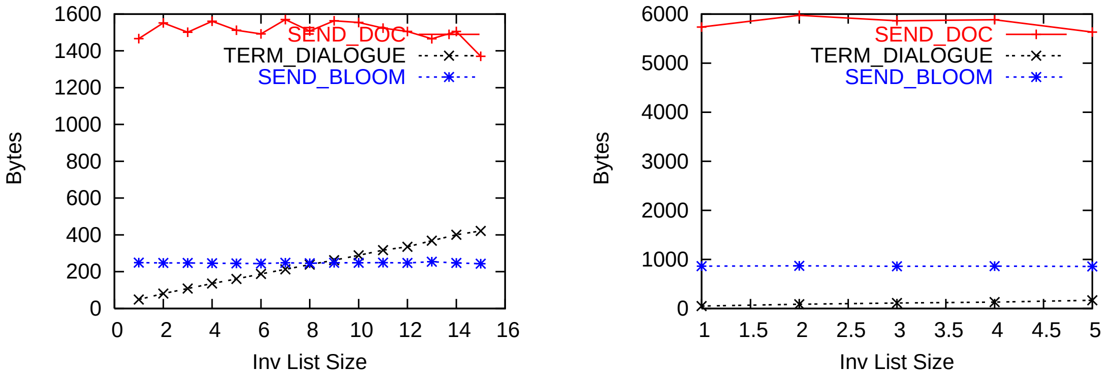


Figure 7: (Uniform Distribution) Bytes Vs Inverted List Size (a) TREC data (b) Citeseer

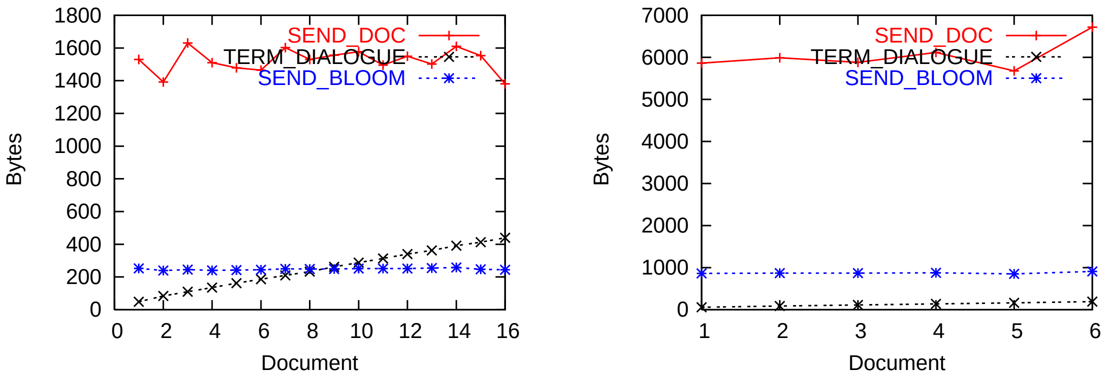


Figure 8: (Inverse skew distribution) Bytes Vs Inverted List Size (a) TREC data (b) Citeseer

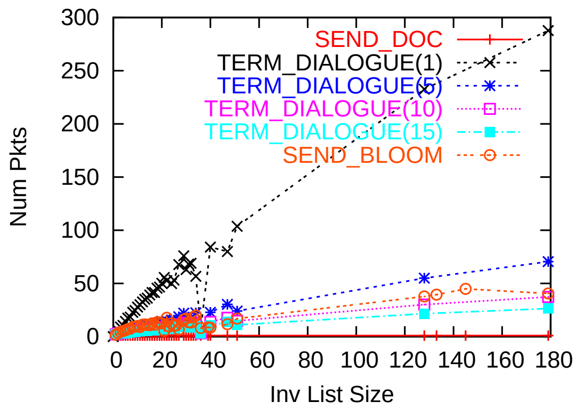


Figure 11: Number of Packets Vs Inv List Size (TREC data)

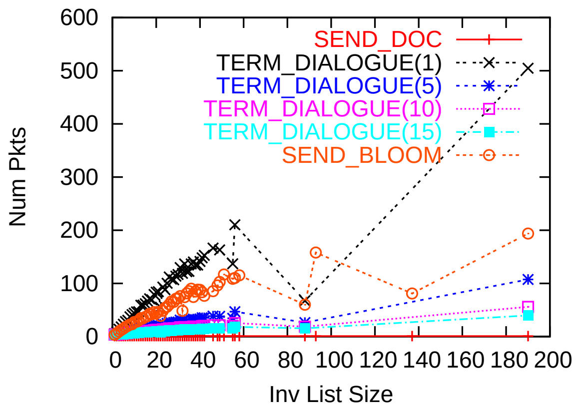


Figure 12: Number of Packets Vs Inv List Size (Citeseer data)

Figure 8. This case is again similar to the uniform distribution: the inverted lists are quite small, and either the bloom filter or the term-by-term dialogue performs best. The inverted lists are small because assuming a ZipF distribution, there are a lot more rare terms, as compared to popular terms, thus the same number of queries are registered on a greater number of distinct terms.

4.6 Notification Type.

We also experimented with varying number of SmartSeer nodes (10, 100 and 1000 nodes) to compare the different notification methods and these results are plotted in Figure 9 (for TREC data) and in Figure 10 (for Citeseer documents). The naive method does not take advantage of multiple terms hashing on to the same query node and thus incurs the same overhead. The clustering method scales linearly (at a low slope) as the number of nodes increases: this can be seen by a simple balls-in-bins analysis. The broadcast method saves lookup cost at low number of nodes, but is clearly wasteful at higher number of nodes. This takeaway here is that the DHT approach clearly helps in reducing the number of nodes contacted as opposed to a mirroring/partition-by-ID method.

4.7 Latency

To measure the latency of the different approaches, we counted the number of packet exchanges for the different optimizations, and the results are plotted in Figure 11 (for the TREC data) and in Figure 12 (for the Citeseer data). The “Send Document” requires only a single round irrespective of the size of the inverted list. The bloom filter method increases slowly with the size of the inverted list, and will probably provide acceptable latency for Smartseer. The term dialogue method requires about one order of magnitude more rounds compared to bloom filter, since terms are requested one after another. This can be remedied by batching terms in groups (the figure shows batches of 5, 10 and 15). These bring down the latency to an acceptable level. Note however, as the batch size increases, the term dialogue has lesser chances to eliminate queries.

4.8 Load Balancing

In order to verify the load balancing properties, we measured statistics related to the amount of processing per node: since the processing is a function of the inverted list sizes, we measured the number of inverted list entries stored at every node that was used in processing the documents. We experimented with a 1000 node system, where each node had 10 virtual servers, so that the keyspace was distributed evenly. Under these conditions, the maximum load on a machine was about 5 times greater than the average load. The deviation from perfect load balancing occurs mainly because of the skewed distribution of queries: this can be addressed by having the number of nodes in a region of the keyspace proportional to the load on the keyspace. Known solutions (cite) can be used to improve this if desired.

5 Public Deployment

In this section we report our initial experience in deploying SmartSeer as a public service.

- Describe how it works together with strib's crawling/incoming document and queries?
- Describe the kind of load we get?
- Any summary performance numbers?

6 Conclusion

In this paper we described SmartSeer, a peer-to-peer preprint repository supporting immediate and continuous keyword queries. We described the basic architecture of the system, and identified several key issues in supporting continuous keyword queries. From our workload simulations we highlight

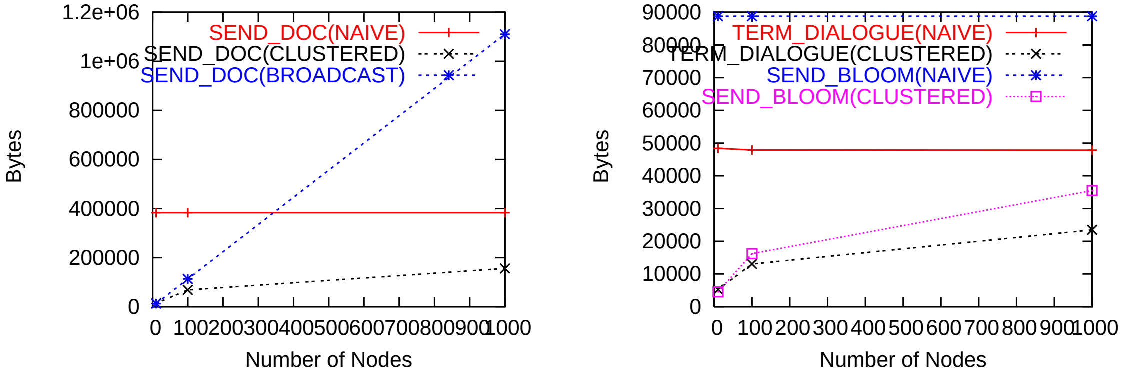


Figure 9: (Varying Number of Nodes: TREC data) Avg Bytes Vs Number of Nodes (a) Send Document (b) Send Bloom Filter and Term Dialogue

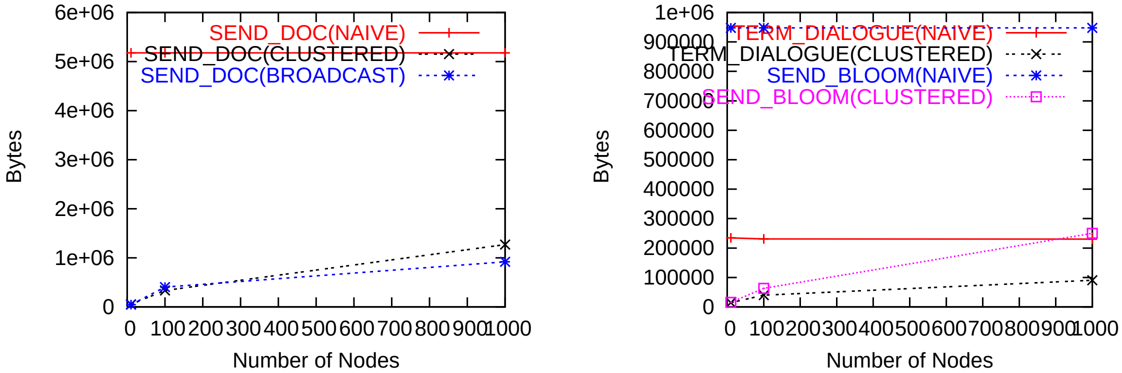


Figure 10: (Varying Number of Nodes: CiteSeer data) Avg Bytes Vs Number of Nodes (a) Send Document (b) Send Bloom Filter and Term Dialogue

the importance of selecting a suitable approach to these challenges. Finally, we report on the challenges arising from a public deployment of the SmartSeer system.

In the future, there are many more aspects of continuous query support that we wish to explore. First, we want to investigate the optimization of more complex queries with subqueries, which can be quite expensive. While the techniques for conjunctive queries may apply, additional optimizations are possible, such as materializing subquery results. Second, we want to explore semantic clustering of documents or terms to further speed up query processing. For example, if frequently co-occurring terms are stored at the same node, then the total number of nodes involved in a continuous query may be decreased. Similarly, if multiple documents are batch-inserted into the system, then some communication between query node and document insertion node (e.g., the resolving of terms) can be "shared" by all documents in the batch. If documents in a batch have similar semantic vectors, then the amount of useful shared communication may increase. Finally, we wish to make use of *relevance feedback* to tune the output of continuous queries.

Acknowledgements. We would like to thank Jeremy Stribling for access to the CiteSeer repository of documents and metadata.