SmartSeer: Using a DHT to Process Continuous Queries Over Peer-To-Peer Networks

Jayanthkumar Kannan*, Beverly Yang†, Scott Shenker‡, Puneet Sharma§, Sujata Banerjee§, Sujoy Basu§, Sung-Ju Lee§
*University of California at Berkeley, kjk@cs.berkeley.edu
†Stanford University, byang@stanford.edu
‡ University of California at Berkeley and International Computer Science Institute, shenker@icsi.berkeley.edu
§ Hewlett-Packard Labs, {puneet,sujata,basus,sjlee}@hpl.hp.com

***Abstract*— As the academic world moves away from physical journals and proceedings towards online document repositories, the ability to efficiently locate work of interest among the torrent of newly-generated papers will become increasingly important. To aid in this endeavor, we designed SmartSeer, a system that allows users to register personalized continuous queries over the CiteSeer database of technical documents. Users are then alerted whenever papers that match their queries are put online.**

SmartSeer has two main design requirements. First, to allow effective information retrieval, it should support rich continuous queries (as opposed to simple keyword searches). Second, to make effective use of donated infrastructure, it should be capable of running on a loosely maintained group of unreliable machines spread across multiple organizations (as opposed to assuming a reliable and tightly coupled distributed system). Existing work on distributed continuous query systems fails at least one of these requirements. Our design for SmartSeer is based on Distributed Hash Tables (DHTs), and thereby leverages previous work on DHT-based query systems. A prototype of SmartSeer has been implemented and evaluated on Planetlab. Though we evaluate our design only for the SmartSeer application, we believe it also provides useful insights into other distributed and rich continuous query systems (web alerts, news alerts etc).

I. Introduction

Academic journals (and their more recent brethren, conference proceedings) have been, for over a century, the primary method for both disseminating and archiving technical documents. The advent of electronic publishing and the web has given rise to various online document repositories that contain not only the electronic versions of journals and proceedings, but also host a much larger collection of documents, including preprints, technical reports, and the like. In some fields, such as theoretical physics, an online preprint collection has largely replaced journals as the primary document repository. Other fields, such as computer science, are moving more slowly, but even in this field the CiteSeer [1] database has become a common way to find technical papers.

As research communities move away from highly filtered venues such as journals and conferences and towards less selective vehicles such as preprint and general document repositories (such as CiteSeer), researchers will find it much harder to keep pace with the literature. The stream of documents flowing into the repository will be far greater than any individual could possibly filter on their own. One way to aid researchers is to allow them to submit continuous (or standing) queries that issue an alert whenever a document matching the query enters the document repository. While certainly not a complete solution to the document-overload problem, such personalized document alerts would greatly reduce the volume of papers individual researchers would have to scan.

This paper describes the design of SmartSeer, a system that allows users to register personalized continuous queries over the CiteSeer database of technical documents. SmartSeer supports instantaneous queries as well (though that is not our primary focus). Our design decisions are also applicable to other large scale event notification systems (*e.g.,* news alerts, web alerts), but in this paper, we only describe how it applies to handling continuous queries over a technical document database. What distinguishes this system from the well-known implementations of continuous queries in traditional databases are two nonstandard design requirements.

The first requirement concerns the nature of queries supported in SmartSeer – to enable more effective information retrieval, we insist that SmartSeer support a family of continuous queries that is far richer than just simple keyword searches. For example, users should be able to search for papers written by their coauthors and for papers that reference their own papers, neither of which are naturally expressed as keyword searches.

The second requirement relates to the infrastructure supporting SmartSeer. We envision eventually deploying SmartSeer as a free public service, much as CiteSeer is now. However, for SmartSeer to scale to significantly more users and documents, as it must if it is to become an important mode of document dissemination, it will require a substantial increase in both the number of hosts and the access bandwidth. Without a revenue stream we cannot foresee how SmartSeer (or CiteSeer) could purchase the necessary resources. However, many universities and other research institutions would likely be willing to donate the use of in-place hosts (and their access bandwidth) to SmartSeer. Thus, we require that SmartSeer should be capable of running on an "opportunistic infrastructure" consisting of donated hosts from multiple organizations. Moreover, far from being a hardened infrastructure, these hosts are loosely maintained and potentially unreliable. These characteristics rules out many traditional distributed information retrieval solutions (such as [2], [3]). We describe the related literature in Section VIII, but for now we note that, to our knowledge,

1-4244-0222-0/06/$20.00 (c)2006 IEEE

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

all the existing work on distributed continuous query systems fails on at least one of the two requirements listed above.

In designing SmartSeer, we face three major design decisions. The first decision is overall system architecture. In Section III, we consider various alternative architectures, such as mirroring queries, and then argue for a design based on Distributed Hash Tables (DHTs). Our approach thus leverages the large body of work on DHT-based query systems, where keyword indexing guides queries to the relevant lists of documents, and extends it to rich continuous queries.

The second design decision is how to best accomplish the rendezvous between queries and documents. In Section IV, we investigate several options – such as "send document to query", "send query to document", and "exchange bloom filter" – in the context of keyword queries and provide a simple analytical model of the performance tradeoffs. In Section V, we investigate these tradeoffs through simulation.

The third design decision is how to support richer queries, such as nested queries and ranking. We discuss these issues in Section VI. We report on our initial implementation in Section VII and conclude with a brief discussion in Section IX.

II. Background

CiteSeer (and other document repositories) currently support *instantaneous* queries, where queries are computed against the current document database, and the answer is returned immediately to the user. Latency is a prime design requirement for instantaneous queries, since a user awaits results from the system. The key scaling factors are the size of the document database and the rate of incoming queries. In contrast, SmartSeer focuses on *continuous* queries: these are standing queries that are applied to each document as it is inserted into the repository. Latency is not a key requirement here (users are not otherwise aware of document insertions, and so are not sensitive to delays in query processing), and the important scaling factors are the number of standing queries and the rate at which documents are inserted. Most online repositories that offer continuous queries are based on a centralized design.

Keyword queries, where users submit a set of *search terms*, are the simplest form of queries we consider. Users can optionally specify a context such as title or author (e.g., *(TITLE:smartseer)* specifies the keyword "smartseer" appearing in the title of the document). Queries are boolean conjunctions or disjunctions of such *terms*, which can be seen as simple predicates expressed using keywords and attribute values. The accepted method of processing instantaneous keyword queries is to use an *inverted index*. An inverted index consists of one *inverted list* per term that appears in the document corpus, where the inverted list consists of the IDs of all documents that contain the given term. Because we process continuous queries, we may use an inverted index over the terms that appear in the queries, rather than in the documents. We refer to the inverted index (list) built over queries as the "query inverted index (list)," and the inverted index (list) built over the documents as the "document inverted index (list)."

Nested queries extend the expressiveness of simple boolean keyword queries by allowing queries on the relationship between entities (e.g., authors, documents, etc.). For example, a query for all documents by co-authors of Jane would be expressed by the nested query *AUTHOR:(AUTHOR:Jane)*. We will discuss such nested queries in Section VI. Most information retrieval systems also support *relevance ranking* of results, where the relevance of a document may be calculated using a number of different metrics. In SmartSeer, we use the standard cosine similarity metric to rank instantaneous query results. For continuous queries, SmartSeer provides the relevance score of individual results to the user.

III. SmartSeer Architecture

We now describe the basic architecture of the SmartSeer system. We note that support for certain kinds of continuous queries requires the ability to support instantaneous queries as well (for more details, refer to Section VI). Thus, the design choice for SmartSeer is dictated by the constraints imposed by both kinds of queries. In terms of performance, SmartSeer should be able to scale to large numbers of registered continuous queries as well as high document arrival rates and should incur low overhead in terms of bandwidth. We begin by comparing different high-level architectures and justifying our choice of a push-based rendezvous based on DHTs. We then describe the details of the SmartSeer implementation over this architecture.

A. Basic Architecture

The highest-level decision to make is between the *pull-based* and *push-based* approaches. In a pull-based architecture, the user polls the system periodically, or the system runs all registered queries as instantaneous queries periodically. Any new results found since the last poll are returned to the user as results to her continuous query. The main disadvantage of such pull-based mechanisms is that a fixed overhead is incurred even if there are no new results. In contrast, in a push-based architecture, costs are only incurred at the insertion of a document. We therefore choose the push-based approach as the more efficient alternative.

There are three well-known design choices for a push-based continuous query architectures: mirroring, partition-by-ID, and partition-by-keyword. In all these architectures, each node maintains a subset of a global index of queries and documents; the difference lies in how these subsets are constructed. In the mirroring approach, all documents and continuous queries are stored on all nodes, and a new document or instantaneous query is sent to a randomly chosen mirror. The partition-by-ID approach partitions the continuous queries (documents) among the nodes, and a new document (instantaneous 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, and this index is partitioned among the different nodes based on the keyword.

We immediately rule out the mirroring option: though it might work well on smaller data sets, it cannot scale to

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

SmartSeer's requirements. For example, the current CiteSeer database has over 795G 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 continuous queries are registered.

Secondly, notice that in the partition-by-keyword method, during the insertion of a new query (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 query (document). We expect that queries will mostly use words that occur in the metadata and abstract of the document (although SmartSeer supports full text search within the document as well). For this common case, the rendezvous approach is more scalable in terms of bandwidth, as the number of nodes that need to be contacted will typically be lower (assuming that the infrastructure consists of the order of hundreds of machines).

For these reasons, we base the design of SmartSeer on the partition-by-keyword architecture. We use a DHT to implement this architecture, thereby allowing us to leverage the existing advantages of DHTs – their ability to run on an unreliable set of nodes, and other useful functionality such as replication, load balancing, etc. Leveraging these advantages considerably simplifies the design of our application.

B. DHT-based design

SmartSeer's design is based on the simple observation (pointed out in PSoup [4]) that the execution of continuous queries can be thought of as running instantaneous 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. Our architecture for supporting continuous queries is a simple extension of the conventional architecture used for executing instantaneous queries over DHTs. For clarity, in this section we focus on the execution of simple conjunctive queries that involve multiple terms of the form *(AUTHOR:name)*.

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 Bamboo DHT [5]. Distributed hash tables provide a simple *put/get* interface that allows insertion and retrieval 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. A term is assigned to the node whose keyspace its hash falls in. We augmented this interface with other SmartSeer-specific operations which we will describe later. We now describe how SmartSeer supports the insertion of new continuous queries and new documents.

Query Insertion: Each conjunctive query is stored at the node responsible for *one* of its terms t_s (assuming for now, that there are no negated predicates), and the query is said to be *registered* at that term. The key of the query is defined to be the hash of t_s. Thus, for conjunctive queries, a query and its metadata are stored only at a single node, in a single inverted list. This approach is different from the document inverted index, where only the document ID (and not the document itself) is stored in an inverted list, and where the ID is stored in the inverted lists of *all* terms in the document.

We choose t_s as the most selective term in the query; doing so allows us 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. In addition, because query inverted lists for popular keywords tend to be larger than for uncommon keywords, registering queries on their common terms would result in higher overhead. Statistics on the selectivity of terms can be obtained in our system by querying the node responsible for storing the *document* inverted list for that term.

Document Insertion: When a new document is inserted, it is parsed into *tokens*, which are then transformed into terms via stop-word filtering and stemming. All nodes responsible for at least one term in the document are then contacted via a *document notification* message. We refer to the node inserting and parsing the document as the *document insertion node* (DN), and each of the nodes contacted as the *query nodes* (QNs). 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. The DN is typically just the node to which a document is initially uploaded. This node is responsible for initiating the protocol for finding matching queries, and also for updating the document indices.

In the basic case, when the DN contacts each QN with respect to a particular keyword, the QN will return the inverted list of all queries registered on that keyword. The DN then matches these queries against the new document, and users are notified of successfully matched queries. We call this approach the *Send Query* method, because queries are shipped between nodes. Note that this method is an exact analogue of query processing in an instantaneous query-processing DHT system.

Note that since instantaneous queries are also supported in our system, we must update the document index on the insertion of a new document, in addition to processing continuous queries. We piggyback these update messages on document notification messages in order to conserve bandwidth. Because a query q is stored on a node responsible for at least one term in q, q is necessarily co-located with the document inverted list of that term. As a result, when a new document is inserted into the system, the nodes that manage the relevant document inverted lists are the same set of nodes that manage the relevant query inverted lists.

C. Scaling Properties

We now analyze the bandwidth consumption of the Send Query method. This analysis is very similar to that for instan-

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

taneous queries in [6]. Reference [6] argued that the bandwidth required to handle instantaneous queries (at the rate of I per second) over N documents is $\beta \times NI \times W_q W_d$, where β is a constant depending on the distribution of terms in queries and documents, and W_q and W_d are the average number of keywords in queries and documents, respectively.

This same kind of analysis may be extended to the case of continuous queries. We assume that the occurrence of keywords follows the same distribution in both queries and documents, and that this distribution is ZipF with parameter α. During the insertion of a document with W_d keywords, the inverted list of queries stored under each keyword is retrieved. The size $Q(r)$ of the inverted list of queries under a given keyword of rank r can be written as $Q(r) = C \times FZ(r, \alpha, W_q)$. Here, $FZ(r, \alpha, W_q)$ is the probability that a query with W_q keywords drawn from the ZipF distribution of parameter α is registered on a keyword of rank r. Assuming each keyword occurs in the document independently, then the retrieval of W_d inverted lists requires bandwidth $W_d \times (\sum_r Z(r, \alpha) \times Q(r))$ where $Z(r, \alpha)$ is the probability that a keyword drawn from a ZipF distribution has rank r. Thus, at a first approximation, the bandwidth required to handle C continuous queries when new documents arrive at a rate R per second, is $\beta \times RC \times FW_d$. Note that most words that occur in documents (such as 'The' etc) will not appear in queries: the effective keyword fraction F accounts for this. This equation is very similar to the equation for instantaneous queries: the number of continuous queries corresponds to the number of documents, and the rate of new documents to the rate of instantaneous queries.

We substituted typical values in this equation: the number of registered continuous queries is set to 1 million queries, the number of words per document was set at $10,000$, the effective keyword fraction was set to 0.01, the average keyword selectivity to 10^{-3}, and the size of a query to 10 bytes. We set the effective keyword fraction by assuming that only words in the abstract of a document will be used to query as search keywords to find that document. This suggests that the communication overhead required to process a single document is about 10 MB. This is about 3 orders of magnitudes smaller than the cost of answering instantaneous queries (which have a very high incoming query rate). Note however, that this cost is still high, and optimizations that cut down on bandwidth consumption are useful. More importantly, this equation shows that the current Send Query approach does not scale with the number of continuous queries: as number of queries C grows, so do the sizes of the query inverted lists, and thus the bandwidth consumption as well. This observation leads us to explore other approaches to "join" query and document data.

IV. Design Alternatives

We now explore three other approaches to join query and document data that are feasible only in the context of continuous query systems. These approaches exploit two main difference between continuous and instantaneous query systems.

First, continuous query systems process inverted lists of queries where the *entire* query is stored in the list. In contrast, an instantaneous query system works with document inverted lists where only the ID of a document is stored in the inverted lists. As a result, rather than shipping the potentially large query inverted list to the document node, each query node can instead process some or all of the queries themselves, and thereby potentially save much bandwidth. Second, a continuous query system has less stringent requirements on latency as compared to an instantaneous query system. This allows us to focus on bandwidth as the main objective, possibly trading off bandwidth with latency (which is still required to be on the order of seconds or minutes).

A. Rendezvous Alternatives

Based on these two observations, we now present three alternatives to the "Send Queries" rendezvous approach.

Send Document: The first design alternative that these differences suggest is the "Send Document" approach, which is a dual of the "Send Query" approach. When a new document is inserted, the *entire* document is sent to all the QNs storing the inverted lists for keywords in the document. This might be beneficial, if for instance, if the query inverted list is much larger than the document itself (for example, if it is a popular keyword). Each QN now matches its stored queries against the new document, and notifies matching queries. This approach becomes useful as the number of continuous queries increases: the bandwidth consumed is at most the cost of sending the entire document to all nodes in the system (it reduces to broadcasting the document).

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

The rationale behind the term-by-term method is that shipping keywords can be much cheaper than shipping entire queries, especially if many queries can be *eliminated* by the resolution of just a few keywords. We say a query is eliminated if, for example, one of its keywords is resolved and found to be absent in the document. Finally, if all terms present in a query are resolved and are known to be present, then the document is known to satisfy the query. Thus, over multiple rounds of requests and response, the QN eventually trims the set of candidate queries to only the successfully matched queries.

There are many opportunities to optimize the term-by-term dialogue. Given general probabilities of the likelihood of terms appearing in documents, a QN may first resolve terms that are not frequent, or terms that appear in many queries, in

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

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. By eliminating queries, as discussed earlier, we may be able to eliminate terms from consideration as well. A QN 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 seems 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 by resolving a different keyword. These are the terms that appear in any remaining query (i.e., a query that has not yet 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. 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.

Bloom Filter: In the "Bloom Filter" method, the DN sends a bloom filter over all terms in the document to each of the QNs. Since the bloom filter has no false negatives, the QN can discard queries that have a term corresponding to a 0 in the bloom filter. In this manner, a QN may prune the set of queries stored in the inverted list to a smaller set. After this step, the QN use the "Send Query" method and send all the remaining queries to the DN. It can alternatively initiate a term-by-term dialogue to process the remaining queries.

B. Analytical Model

In this section, 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 (Section V), we demonstrate the utility of this model in determining the best approach. 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).

We note that the Term Dialogue approach is always superior to the Send Queries method, since 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. Thus, we will only consider the other three approaches: Send Document, Term-by-Term Dialogue, and Send Bloom Filter. These three approaches can all be optimal in different scenarios. The bandwidth cost SD of shipping a document is simply d, 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, q, 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. In the extreme case, if every query contains the term t which is found not to be present, then all queries are eliminated, and the cost of the dialogue is simply the size of two small messages. Thus, the actual cost of the dialogue is $TD = \delta \cdot q$, where δ is a fudge factor depending on which terms are actually present in the document, and the order in which terms are resolved.

The bandwidth cost BF of shipping a bloom filter, followed by a term-by-term dialogue for all non-filtered query terms, is $BF = f(d) + \delta' Q'$. Here, $f(d)$ is a function determining the "optimal" bloom filter size given a document size and Q' is the number of *queries* at the QN after the bloom filter has been used to prune the list of queries. The term $\delta' Q'$ accounts for the cost of the term-dialogue once the the bloom filter has been applied. In order to calculate Q', first let us define Q_i as the number of queries such that all but i of their terms appear in the document. Thus, Q_0 is the number of the true matching queries, and since a bloom filter has no false negatives, Q' includes these queries. Q' also includes some false positives whose number can be expressed using the false positive rate f of the bloom filter:

$$Q' = Q_0 + \sum_{i=1}^{i=k} Q_i f^i$$

The second term accounts for all the false positives. Denote the total number of queries at the QN, before the bloom filter was applied, as Q (the number of *query terms* q is νQ where ν is the average number of terms per query). Then, ignoring the higher order error terms $Q_2, Q_3, \cdots$ (assuming we have a Bloom Filter with a low false positive ratio):

$$BF = f(d) + \delta'(\gamma_0 Q + f\gamma_1 Q) = f(d) + \gamma q + \epsilon q$$

where Q_0, Q_1 are written as $\gamma_0 Q, \gamma_1 Q$. The second equation is obtained by suitably defining the constant terms γ, ϵ. The term ϵ is proportional to the error probability of the bloom filter and the selectivity of the document, while the term γ is proportional to the selectivity alone. The size of the optimal bloom filter can be written as a function of the required error probability ϵ.

Looking at the above equations, the basic tradeoffs between the three approaches is clear. If d is large relative to q, then Term 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, Send 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 Bloom Filter is better than Send 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 Send Document is best. Later in Section V, we will quantitatively compare these approaches through simulations of SmartSeer over typical

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

document workloads, and show how the above simple model predicts well the best approach to use.

Recall that in all approaches, the document insertion node must first send a document notification message to all QNs corresponding to all terms in a new document, to alert them of the need to process continuous queries over this document. In this message, we can include the bloom filter, document, or any other information necessary to implement the desired join approach.

C. Batch Notification

In addition to considering different rendezvous approaches, we must also be careful of how each approach is implemented. In this section, we study how we might *batch* process document notifications by node, and when doing so is beneficial.

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 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 assumed thus far. To illustrate, say the terms “john”, “james”, and “jack” all hash to the same node, and that the Send Document approach is in use. As currently described, the DN will send the document to the same node three times, which is expensive. Alternatively, by recognizing that the above three terms hash to a single node, the DN can send the document to that node just once.

Therefore, in the *clustered* approach to document notification, the document insertion node will first find, for each term, the QN responsible for 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. Note however that this clustered approach is complex, and difficult 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. In addition to being complex and error-prone, the clustered approach has a high latency due to the processing required to look up all necessary mappings. Therefore, the clustered approach is feasible when the number of nodes in the system is relatively small.

If the number of nodes in the system is very small compared to the number of unique terms in a document, 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 (e.g., [7]). However, broadcast cannot be used for all join approaches. In particular, it cannot be paired with a term-by-term dialogue, since a QN receiving the notification will have no idea which terms in its keyspace are relevant to the document. Similarly, broadcast cannot be paired with bloom filters. Broadcast is only appropriate in conjunction with the Send Document approach. With the full document, the QNs can extract which terms in its keyspace are relevant to the document, and thereby update the appropriate inverted lists and process the appropriate queries.

TABLE I
DEFAULT PARAMETER VALUES FOR SIMULATION.

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

V. EVALUATION

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

Our SmartSeer implementation is written in Java using the libraries exported by Bamboo and OpenDHT [8]. Since in our system, the node storing the tuples may have to do complex operations other than put/get (such as sending bloom filters), we use the ReDiR OpenDHT library that stores tuples on application hosts rather than OpenDHT hosts. We use this same implementation both for deployment (for more details, refer to Section VII), and for simulating multiple nodes.

We run SmartSeer over two document sets: CiteSeer [1], and TREC [9]. 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 other 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) 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 a workload of synthetically generated queries. There are few operational continuous queries systems whose query logs are widely available, so we rely on synthetic queries whose 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

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

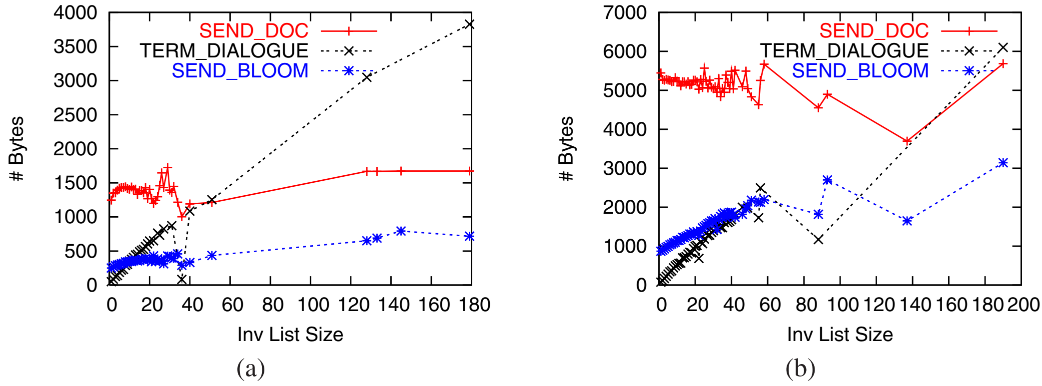

Fig. 1. Bytes Vs Inverted List Size (a) TREC data (b) CiteSeer

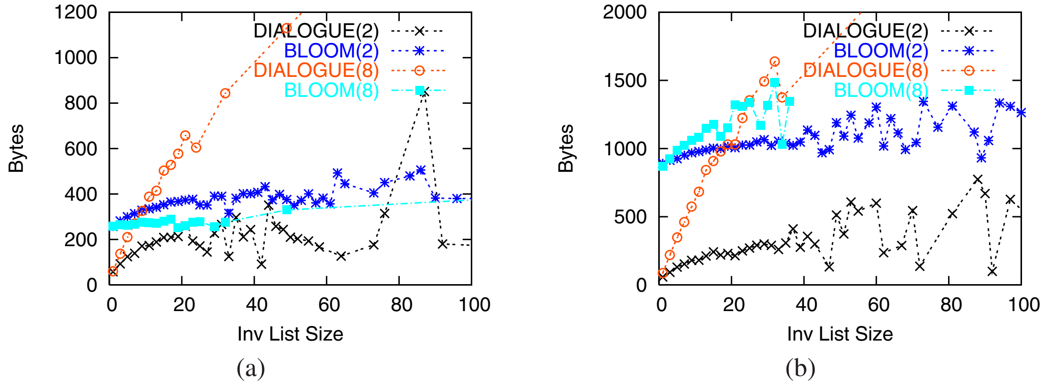

Fig. 2. (Varying Query Selectivity) Bytes Vs Inverted List Size (a) TREC data (b) CiteSeer data

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 distribution, and the next batch is used to measure the bandwidth consumed. The above description of 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 I.

A. Basic Comparison

First, we compare the three join approaches under different workload scenarios. Figure 1 shows the average number of bytes exchanged between the DN and a single QN on the insertion of a new document. Because different QNs may have different loads, as some QNs may be responsible for large inverted lists, along the x-axis we vary the size of the inverted lists, where size is measured by the number of queries in the list. We show one curve for each of the three join approaches.

In this figure we see clearly the basic tradeoffs described in Section IV. First, note that the term q used in our analysis is (approximately) proportional to the inverted list size. When the inverted list size (and thus, q) is low, Term Dialogue has best performance. As inverted list size and q grow, however, the cost of Term Dialogue grows linearly with q. Likewise, the cost of the Bloom Filter approach grows linearly with q; however, its slope $(\epsilon + \gamma)$ is so small, this approach still has good performance even when q is fairly large. The overhead of the Send Document is nearly constant since the complete document is sent. Note that documents are compressed (using gzip) before being sent on the network. In our simulations under the default parameters, we did not reach the point where Bloom Filter has worse performance than Send Document.

B. Query Selectivity

Recall from Section IV that Bloom Filter performs relatively poorly when γ is large – that is, when queries are unselective and thus many queries are satisfied by the document. Intuitively, the more selective a query, the more likely the query will be eliminated by the bloom filter. To study the impact of query selectivity on performance of the join approaches, we indirectly tune selectivity by varying the average number of terms per query – the higher the number of terms, the more selective the query.

Figure 2 shows us the performance of Term Dialogue and Bloom Filter in terms of the number of bytes exchanged between the DN and a single QN, again as we vary inverted list size along the x-axis. We do not plot the performance of Send Document since it is constant irrespective of the selectivity of the queries. For join approach we show two curves: one representing the case where queries have an average of 2 terms, and one where queries have an average of 8 terms.

From this figure, we see that as the number of terms per query increases (and thus the queries become more selective),

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

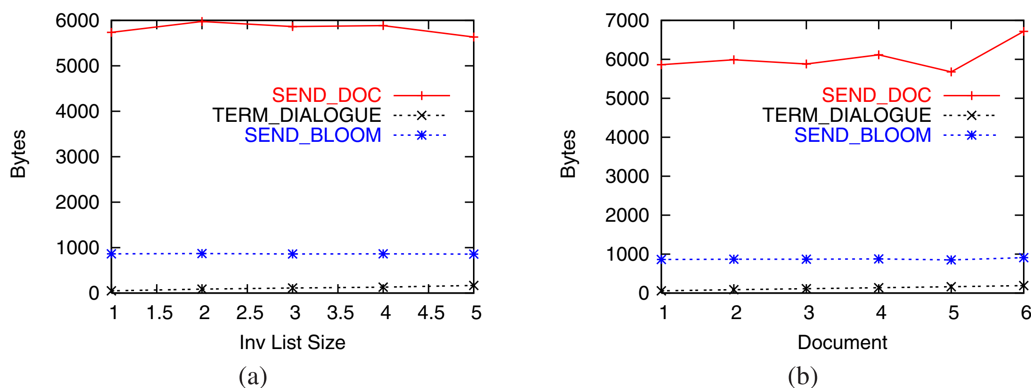

Fig. 3. (Varying Term Distribution: CiteSeer data) Bytes Vs Inverted List Size (a) Uniform Distribution (b) Inverse Skew Distribution

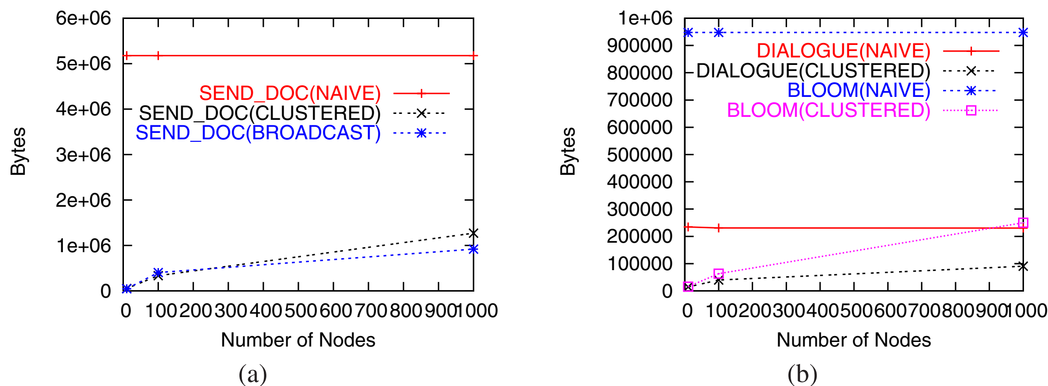

Fig. 4. (Varying Number of Nodes: CiteSeer data) Avg Bytes Vs Number of Nodes (a) Send Document (b) Send Bloom Filter and Term Dialogue

the performance of Bloom Filter improves relative to Term Dialogue. For example in Figure 2a (over TREC data), when inverted list size is 40 and queries have an average of 8 terms, Bloom Filter outperforms Term Dialogue by over a factor of 3. When the number of terms is very low, however, Term Dialogue outperforms Bloom Filter, because of highly unselective queries. In Figure 2a Term Dialogue consistently outperforms Bloom Filter by a factor of two when queries have an average of 2 terms; in Figure 2b (over CiteSeer data), Term Dialogue outperforms Bloom Filter by over a factor 3.

Therefore, when queries are unselective – such as when they contain few keywords, or when they contain common keywords – Term Dialogue is preferred over Bloom Filter. In addition, recall from Figure 1 that when q is large (i.e., when the inverted list sizes are large), Send Document outperforms Term Dialogue. Thus, Send Document is the optimal approach when many unselective continuous queries are registered.

C. Distribution of Terms

Recall from Section V that by default, the popularity distribution of query terms is set to be the same as the distribution of document terms. In addition, we can perturb the data by increasing the skew of the distribution, ignoring the skew (i.e., using a uniform distribution), and inverting the skew. In Figure 3 we show the performance of our three join approaches under the (a) uniform and (b) inverse skew distributions. Figure 1 already studied the default skew, and we omit increased skew results as they are similar to default skew results. Figure 3 only displays results over CiteSeer data; experiments over TREC data yielded similar results.

From Figure 3, we find that most inverted lists are very short: the maximum size being about 6. Such conditions favor Term Dialogue over other approaches, as observed earlier in Figure 1. The reason for short inverted lists is that that under the uniform and inverse skew distributions, the domain of terms on which queries are registered is very large. In contrast, under the default or high skew distributions, queries mostly only contain common terms, which leads to fewer and larger inverted lists. As a result, the distribution of query terms affects the inverted list size, which in turn influences the tradeoffs between the join approaches.

D. Batch Notification

Figure 4 shows the performance of the different notification methods (Naive, Clustered, and Broadcast) described in Section IV-C, as the number of (simulated) nodes in the system is varied along the x-axis. In this figure, performance is measured as the total number of exchanged bytes across *all* query nodes, on average, across document insertions.

First, note that for all three join approaches, the Naive notification method, which does not batch notifications by node, has the same performance regardless of number of nodes. Clearly, then, performance is suboptimal when the number of nodes is small. For the Send Document approach,

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

the bandwidth for Broadcast grows linearly with the size of the system, as would be expected. The Clustered notification method also grows roughly linearly with system size: however, note that with Clustered, the Bloom Filter method has a much higher slope than the Term Dialogue method. The reason for this observation is that as the number of nodes increases, the size of the inverted list at each node decreases, and as we have seen before, Term Dialogue outperforms Bloom Filter when the inverted lists are small. Also note that as the number of nodes increases, the performance of Clustered notification approaches that of Naive.

Given that Clustered notification always outperforms (or performs similarly to) Naive in terms of bandwidth, we might be tempted to conclude that Clustered is always preferred over Naive. However, not shown in this figure is *latency*, which is a small constant for Naive, but is roughly linear to min(number of nodes, number of distinct terms in document) for the Clustered approach. If the number of nodes is large enough such that the bandwidth difference between these approaches is small, the Naive method is preferred for superior latency, as well as improved robustness and simplicity, as discussed in Section IV-C. Note also that for the Send Document approach, the Broadcast and Clustered methods perform similarly; therefore, again due to simplicity, latency and robustness, the former is preferred.

E. Discussion

Our experiments in this section clearly demonstrate the tradeoffs between different join and notification approaches, which we found to be highly dependent on a number of parameters: query selectivity, size of inverted lists, etc. An optimal continuous query system, therefore, cannot simply implement a single approach, unless all workload characteristics are known at design time and do not evolve. Instead, we believe our experiments make it clear that an *adaptive* approach is best: one in which all approaches are implemented, and the best approach is chosen according to current workload.

For example, when our SmartSeer application is initially deployed with few nodes and few registered queries, we may use the Term Dialogue join approach with Clustered notification. As the number of continuous queries and nodes grow, we may switch to Bloom Filter. However, if inverted lists become extremely large and we find queries to be very unselective, we may revert to Term Dialogue or even Send Document, and use Naive or Broadcast notification. Note that approaches can also be determined on a case-by-case basis; for example, if only a short abstract is submitted, the DN may unilaterally select the Send Document join approach, and send the small abstract to all relevant nodes. Such decisions can be made by monitoring selectivity and other statistics, which can then plugged into the analytical model discussed earlier.

VI. Handling complex queries

Our discussion thus far has been limited to simple conjunctive queries with all predicates based on equality. In this section, we describe how to extend our techniques to the following classes of queries:

- **Queries with arbitrary boolean expressions:** These are queries with the basic equality predicates composed using the operations AND, OR, and negation (!).
- **Nested Queries:** Nested queries are queries whose terms can contain subqueries and predicates using the relation IN (membership function). *e.g., Q1 = AUTHOR:IN:Q2* where *Q2* is subquery that returns a list of authors. Note that we generally omit the *IN* keyword, and would write the above query as *Q1 = AUTHOR:Q2*

We also note that there are certain categories of queries that SmartSeer does not support efficiently because of limitations imposed by the DHT-based rendezvous mechanism: we discuss such limitations at the end of this section.

A. Boolean Queries

We first describe how queries with negated predicates and OR queries (queries where the boolean expression is a disjunction of simple predicates) are supported, and then explain how to handle arbitrary boolean expressions.

Negated Predicates: Consider a query Q which has negated predicates (of the form $!T$ where T is *Attribute:Value*). Such a query can only be registered on its *non-negated* predicates efficiently, since a DHT only directly supports registration based on equality. In a DHT, the only way to register a query on a negated predicate of the form *!(A:V)* would be to register the query on all terms *A:V'* (such that $V' \neq V$), which would be clearly inefficient.

Thus, SmartSeer always registers such a query on its non-negated predicates. Negated predicates are matched during the process of joining the query and the document. For example, consider a conjunctive query *Q = (!S) AND (T)* , registered on non-negated term T. In the Send Document method, the QN responsible for T ensures that Q is matched only if the document satisfies the negated predicate $!S$ as well (i.e., by verifying that the document does not contain S). The other join approaches handle negated predicates in a similar fashion.

Note that the approach taken for negated predicates can also be taken for other predicates that are not supported over a DHT. For example, SmartSeer is unable to process range predicates, as DHTs only support equality. If a conjunctive query contains a range predicate as well as non-negated equality predicates, SmartSeer will simply register the query on a non-negated equality predicate. Later in Section VI-C we discuss the case where certain combinations of difficult predicates lead to a query that SmartSeer cannot support.

OR Queries: Consider OR queries of the form $T_1 \; OR \; T_2 \; \cdots \; OR \; T_k$. OR queries are registered on every term, rather than the least selective (or any single term). 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, however, the query may be processed multiple times (duplicate processing). One possible optimization is to rewrite an OR query into

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

multiple queries such that only one of them is matched. A query of the form $Q = (T_1 \ OR \ T_2 \ OR \ \cdots \ OR \ T_k)$ can be written into k queries of the form $Q_i = (!T_1) \ AND \ (!T_2) \ AND \ \cdots \ AND \ (!T_{i-1}) \ AND \ T_i$. Each of these queries Q_i has only one positive term, T_i, on which it is registered. It is clear that a new document cannot match more than one out of these set of queries: this method thus minimizes wasteful processing.

Boolean Queries: An arbitrary boolean query Q can be decomposed into its disjunctive normal form (DNF) where Q is written as $Q_1 \ OR \ Q_2 \ \cdots \ OR \ Q_k$, and each clause Q_i is an AND query. We then register each clause as we would an independent AND query, and on a match to any of these queries, the owner of the parent query Q is notified.

B. Nested queries

Technical documents have rich meta-data associated with them (such as list of authors, list of citations etc), that imply a relationship between different entities (*e.g.,* a paper *is written* by its author, a paper *is cited* by another). So far, we have allowed the user to query based on first-order constraints, such as a paper written by XYZ, or a paper that cites paper X. Nested queries are queries whose terms involve subqueries, and such queries allow users to express *higher-order* constraints: *e.g.,* papers *cited by* papers *that cite* paper XXX. SmartSeer supports such nested queries by translating the complex query into multiple subqueries and registering these subqueries. Note that traditional database joins may be expressed using nested queries.

To motivate the need for such nested queries, we list some sample queries below that we expect in SmartSeer, outside of simple keyword queries. First, users of the system might be interested in tracking citations to their papers. The naive way of stating this query would be to insert the query *(CITATION:*X_1 *OR CITATION:*X_2 *OR ....OR CITATION:*X_n*)*, where $X_1 ... X_n$ are the IDs of all the user's papers. Alternatively, and more conveniently, we could state the following nested query: *CITATION:IN:(AUTHOR:author_name)*. In other words, return papers that contain a citation in the set of papers written by *author_name*. As another example, consider the user John who is interested of keeping track of new papers written by his co-authors. The query *(AUTHOR:IN:(AUTHOR:John))* would notify John of papers written by people who are in the set of authors found in all papers written by John (i.e., John's co-authors). Since nested queries allow users to concisely express their interests, SmartSeer supports such queries both as continuous queries and instantaneous queries. For clarity, we first discuss how SmartSeer handles nested instantaneous queries, and then show how to extend this mechanism to handle continuous queries.

1) Nested Instantaneous Queries: When a query contains a subquery as a term, we first execute the subquery to retrieve a list of documents D that satisfy the subquery. From such matching documents, attributes 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 is then used in processing the original query.

For example, say a user submits the query Q = *(YEAR:2004 AND AUTHOR:(networks AND AUTHOR:Bob))* in order to find papers written in 2004 by authors who co-authored a paper with Bob containing the text 'networks'. SmartSeer will execute the subquery (networks AND AUTHOR:Bob) to find all documents $D = \{X_1, X_2\}$ that satisfy the subquery. Say documents X_1 and X_2 are written by Bob, AuthorX, AuthorY and AuthorZ. Then, the translated version of this subquery is *(AUTHOR:AuthorX OR AUTHOR:AuthorY OR AUTHOR:AuthorZ)* . Clearly, given the state of the database (at the time the query is first submitted), this translated subquery is equivalent to the original subquery term *AUTHOR:(networks AND AUTHOR:Bob)*. SmartSeer then incorporates the translated subquery into its parent query Q to form the translated query Q' = *(YEAR:2004 AND (AUTHOR:AuthorX OR AUTHOR:AuthorY OR AUTHOR:AuthorZ))*. Q' is now a normal boolean query, and can be executed as described earlier.

2) Nested Continuous Queries: Continuous nested queries are slightly more complicated. First, when the query Q is first registered, the subquery is translated as described earlier. Once all subqueries have been translated, the translated query Q' is registered as a normal boolean query. In addition, the original subqueries themselves are also registered.

Consider our example from the previous section *(YEAR:2004 AND AUTHOR:(networks AND AUTHOR:Bob))*. The original subquery is *(networks AND 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. This document will match the translated query, and will thus be returned as a continuous query result. Note that if the query were not registered in its translated form, then for every document inserted, we would have to re-evaluate the entire subquery and query expression as described in the previous section. Therefore, the translated subquery is essentially a "materialized view" of the results of the subquery, necessary only to improve efficiency.

However, registering the translated subquery is not sufficient for correctness. Say Carol has never written a paper with Bob before on networks, until now. When their new document is inserted, Carol now satisfies the original subquery, and suddenly all of Carol's documents from 2004 now satisfy the query. Therefore, when the original subquery is matched by this new paper with Bob, the translated query is rewritten as *(YEAR:2004 AND (AUTHOR:Carol AND (!AUTHOR:AuthorX) AND (!AUTHOR:AuthorY) AND (!AUTHOR:AuthorX))* , and this rewritten version is executed as an instantaneous query. Note that to ensure that only new results are returned (i.e., only those papers by Carol), the rewriting states that the author is not any one of the previous authors. Note that these mechanisms can be extended in a straightforward way to support multiple levels of nesting.

SmartSeer does not allow negated subqueries in a con-

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

tinuous query since, as noted in previous literature, such a subquery could require that results for a continuous query be "recalled." As there is no clean solution to this problem, SmartSeer disallows negated terms in continuous complex queries.

C. Limitations on Expressiveness

There are certain classes of queries that SmartSeer does not support efficiently because of limitations imposed by our implementation of the DHT-based rendezvous mechanism.

First, SmartSeer does not currently allow *similarity* queries that ask for all documents similar to a given document. We handle such similarity queries by assuming that the user is willing to augment such queries with a few keywords. We then register the query on these keywords, and when a new document is inserted containing these keywords, we can use our ranking mechanism to determine whether the document is above a certain threshold of similarity with the query. We can also allow the user to vary this threshold dynamically, so that the system can utilize feedback from the user. We leave examination of such *relevance feedback* for future work.

Second, there are some restrictions regarding the *structure* of the queries allowed by SmartSeer, when difficult predicates (negation, range) are present. For example, queries with all negated terms cannot be handled in our system; the rendezvous mechanism requires at least one non-negated term on which to register. To determine whether a query can be supported by SmartSeer, we describe the following simple algorithm: Express the query as a parse tree, where leaves are simple term predicates, and internal nodes are clauses – an AND node is a conjunctive clause, an OR node is a disjunctive clause. Mark all non-negated equality term predicates. Then, recursively mark any internal node if (1) it is an AND node and at least one child is marked, or (2) it is an OR node and all children are marked. If the root node is marked, SmartSeer can support the query; otherwise not.

For queries that SmartSeer cannot directly support, one possible action is to store these queries on a dedicated set of machines. This set of machines will receive *all* new documents as they are inserted, over which they will process all the "unsupportable queries." In practice we expect this solution to work well, as SmartSeer should be able to handle almost any meaningful query in this domain.

VII. Deployment

We have a basic implementation of SmartSeer running on Planetlab currently and in this section, we report the performance of our preliminary SmartSeer deployment. The purpose of this initial rollout is to debug our system as well as to obtain performance numbers that validate our design. In this initial deployment, 20 machines were chosen from different Planetlab sites and the average TCP throughput between two machines was about $10 - 15$ Mbps.

In these experiments, we used a query corpus of about hundred thousand queries and a document corpus of 1000 documents. Continuous query logs for the CiteSeer database were not available, so we converted instantaneous queries to CiteSeer (from a MIT log) into continuous queries. All queries are simple conjunctive queries (although our implementation supports complex nested queries as well). Out of these $96,000$ distinct queries, about $4,000$ of them are from real MIT CSAIL (Computer Science and AI laboratory) users, and the remaining are from search engine crawls (queries such as "author year title" obtained from web page links). Since most of these queries are likely to refer to papers in recent years, we chose the most recent documents available to us from the CiteSeer database. The document corpus consists of 1000 documents randomly chosen from about 4000 documents that were inserted in the first half of 1999 into CiteSeer (we could not get access to more recent documents).

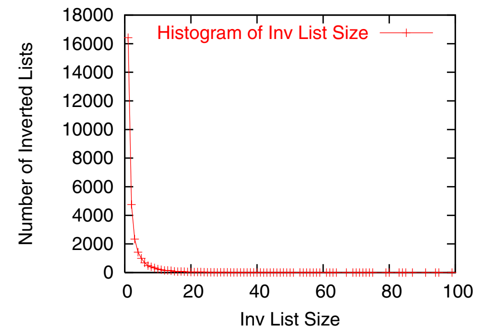

Fig. 5. Distribution of Inverted List Sizes

Our methodology was as follows: we inserted 100 documents into the system, followed by the entire query corpus, and then the entire document corpus. The initial set of documents are inserted so the system can obtain selectivity of keywords (which is used in registering the queries). After all documents and queries have been inserted, we measured the throughput of the system by observing the time taken to finish processing all 1000 documents. The time taken to insert a new continuous query is very low (of the order of milli-seconds), so we do not report those results in detail. The bandwidth consumption was already extensively studied in simulations using the same code, so we do not report them either. We tested all the three join approaches: Send Document, Term Dialogue, and Bloom Filter. Since the number of nodes in the system is low (compared to the average number of terms in a document, which is roughly 2000), we used Clustered notification.

In order to understand some of the query characteristics, we first plot the size of the inverted lists in Figure 5. This plot suggests that most inverted lists have very few queries, while some of them can get very long: this kind of heavy-tail distribution is expected. In fact, there are a few inverted lists with sizes of a few thousand, but these are not shown in the graph. Note that this distribution is influenced by the fact that SmartSeer chooses the most selective keyword to register a query: for this reason, this distribution is a slightly normalized version of the underlying query distribution.

The results of our experimental study on the throughput of the system are as follows. The Send Document, Term

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.

Dialogue, and Bloom Filter approaches can support $66,886$, $78,728$, and $81,430$ document insertions per day. The Bloom Filter and Term Dialogue methods provide nearly the same performance, while Send Document is the costliest. The throughput of this system is about $80,000$ new documents per day, which is probably adequate for a preprint library. Therefore, we expect that our design can easily match the performance requirements for a preprint library. Note, however, that the number of machines and number of registered queries are likely to be much higher in a realistic public deployment.

VIII. Related Work

Related work can be organized into into four main categories. First, there is a large body of work on distributed database solutions (eg: Mariposa [10]) that provide strict ACID semantics and operate on a set of reliable machines. Such systems are not designed for an opportunistic infrastructure, which is one of the key requirements of SmartSeer.

Second, much research has been conducted on supporting continuous queries in a centralized setting (*e.g.,* Eddies [11], Babcock et. al [12]). These systems support and optimize complex continuous queries, though clearly, many of these techniques do not apply in the setting of a distributed opportunistic infrastructure. For example, they do not deal with the possibility of failure during an ongoing computation. We also note that CiteSeer [1] itself supports continuous queries using a centralized database (it is however not available to users: presumably, because of heavy load). Clearly, such an approach does not scale as the number of queries increases.

The third category consists of more recent solutions from the database world for supporting instantaneous queries over DHTs. Examples of such systems are PIER [13] and Li et. al. [6] (which studied the feasibility of indexing the web using DHTs). PIER is a general system that is designed for arbitrary schemas and arbitrary queries. It also supports continuous queries to a limited extent (though the details of supporting complex queries are not specified). PIER treats continuous queries as long-running instantaneous queries, and thus processes even related queries (queries sharing a keyword) as individual queries, which can be potentially expensive. On the other hand, SmartSeer is meant to support continuous queries, and the design is optimized towards such a workload. Other works such as Overcite [14] and Felber et. al. [15] support instantaneous queries over a CiteSeer-like document database. There is a rich literature on supporting instantaneous queries, and the design of SmartSeer leverages on such efforts.

Finally, existing work has studied continuous queries over distributed systems. Systems such as Scribe [16] provide event notification systems over a distributed architecture; however, they are mainly concerned with supporting simple keyword queries. There has been considerable research in supporting continuous XML queries which can be very expressive (eg: Onyx [17]), but such research focuses on building a network for routing queries and documents, and does not address unreliable infrastructure.

IX. Conclusion and Future work

In this paper we described SmartSeer, a distributed preprint repository supporting rich continuous queries. We described the basic architecture of the system, and identified several key issues in supporting continuous keyword queries. We point out key differences between continuous and instantaneous queries that allow us to implement the joining of documents and queries by new techniques which are not feasible in the latter scenario. From our simulations, we highlight the importance of selecting a suitable approach to these challenges. SmartSeer also handles a rich set of complex queries such as nested queries, by rewriting them into simpler queries. Finally, we report on some preliminary performance results from a deployment of the SmartSeer system. In the future, we wish to explore adaptive join approaches based on the size of the document and the inverted lists. We are also interested in evaluating and optimizing the performance of nested queries.

Acknowledgment

We would like to thank Jeremy Stribling for access to the CiteSeer repository of documents and query logs from MIT.

References

[1] K. Bollacker, S. Lawrence, and C. L. Giles, “A system for automatic personalized tracking of scientific literature on the web,” in *ACM Digital Libraries*, New York, 1999.
[2] B. Ribeiro-Neto and R. Barbosa, “Query performance for tightly coupled distributed digital libraries,” in *ACM Digital Libraries*, June 1998.
[3] B. Ribeiro-Neto, e. S. Moura, M. S. Neubert, and N. Ziviani, “Efficient distributed algorithms to build inverted files,” in *ACM Conference on R&D in Information Retrieval*, August 1999.
[4] S. Chandrasekaran and M. J. Franklin, “PSoup: a system for streaming queries over streaming data,” *VLDB*, vol. 12, no. 2, pp. 140–156, 2003.
[5] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, “Handling churn in a DHT,” in *USENIX Annual Technical Conference*, Boston, June 2004.
[6] J. Li, B. Loo, J. Hellerstein, F. Kaashoek, D. Karger, and R. Morris, “The feasibility of peer-to-peer web indexing and search,” in *International Workshop on Peer-to-Peer Systems*, Berkeley, California, June 2003.
[7] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, “Splitstream: High-bandwidth content distribution in a cooperative environment,” in *IPTPS*, Feb. 2003.
[8] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu, “Opendht: A public dht service and its uses,” in *SIGCOMM*, 2005.
[9] “TREC data,” trec.nist.gov/data.html.
[10] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu, “Mariposa: A wide-area distributed database system,” *The VLDB Journal*, vol. 5, no. 1, pp. 048–063, 1996.
[11] R. Avnur and J. Hellerstein, “Eddies: Continuously adaptive query processing,” in *ACM SIGMOD*, May 2000.
[12] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and Issues in Data Stream Systems,” in *ACM Symposium on Principles of Database Systems*, June 2002.
[13] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica, “Querying the internet with pier,” in *VLDB*, Sep 2003.
[14] J. Stribling, I. G. Councill, J. Li, M. F. Kaashoek, D. R. Karger, R. Morris, and S. Shenker, “Overcite: A cooperative digital research library,” in *IPTPS*, Feb 2005.
[15] P. Felber, E. Biersack, L. Garces-Erice, K. Ross, and G. Urvoy-Keller, “Data indexing and querying in dht peer-to-peer networks,” in *ICDCS*, 2004.
[16] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, “SCRIBE: A large-scale and decentralized application-level multicast infrastructure,” *IEEE Journal on Selected Areas in communications*, 2002.
[17] Y. Diao, S. Rizvi, and M. J. Franklin, “Towards an internet-scale xml dissemination service,” in *VLDB*, Toronto, August 2004.

This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the Proceedings IEEE Infocom.
Authorized licensed use limited to: International Computer Science Inst (ICSI). Downloaded on July 15,2010 at 00:25:54 UTC from IEEE Xplore. Restrictions apply.