Move: A Large Scale Keyword-based Content Filtering and Dissemination System

Weixiong Rao* Lei Chen† Pan Hui‡ Sasu Tarkoma*
*Department of Computer Science
University of Helsinki, Finland
{weixiong.rao, sasu.tarkoma}@cs.helsinki.fi
†Department of Computer Science and Engineering
Hong Kong University of Science and Technology
leichen@cse.ust.hk
‡Deutsche Telekom Laboratories
Germany
pan.hui@telekom.de

Abstract—**The Web 2.0 era is characterized by the emergence of a very large amount of live content. A real time and fine-grained content filtering approach can precisely keep users up-to-date the information that they are interested. The key of the approach is to offer a scalable match algorithm. One might treat the content match as a special kind of content search, and resort to the classic algorithm [5]. However, due to blind flooding, [5] cannot be simply adapted for scalable content match. To increase the throughput of scalable match, we propose an adaptive approach to allocate (i.e, replicate and partition) filters. The allocation is based on our observation on real datasets: most users prefer to use short queries, consisting of around 2-3 terms per query, and web content typically contains tens and even thousands of terms per article. Thus, by reducing the number of processed documents, we can reduce the latency of matching large articles with filters, and have chance to achieve higher throughput. We implement our approach on an open source project, Apache Cassandra. The experiment with real datasets shows that our approach can achieve around folds of better throughput than two counterpart state-of-the-arts solutions.**

I. Introduction

The Web 2.0 era is characterized by the emergence of live content, such as breaking news on the Internet, user generated content on social networks, RSS streams. These live content items have a common characteristic, very large scale. For example, on Facebook, there are 1.5 million wall posts, 2.7 million photos, and 10.2 million comments in only 20 minutes [2]. These numbers indicate a shift from small scale static web pages to large scale real time content. The changes imply new requirements from users to access content of interest.

First with real time content, users would like to be timely alerted of any updates. Otherwise, the users have to tediously enter input keywords to search fresh content of interest. Second, with a very large amount of content, it is quite necessary to use a fine-grained filtering model to filter out irrelevant content. Unfortunately, most current web sites offer coarse filtering models. For example, Facebook simply allows a user to either follow other users' activity to receive all information of such users, or block such users without receiving any information. Similarly, My Yahoo! and iGoogle allow users either to subscribe selected RSS streams or to completely unsubscribe the RSS streams. However, in many case, users are interested in only some relevant postings of the followed users (or RSS streams), and want to filter out all other postings. Thus, the current web sites cannot offer fine-grained filtering.

Driven by these requirements, we propose to use the push-based content filtering and dissemination manner, such that users can be timely alerted of fresh content. Next, we adopt a keyword-based model to offer fine-grained filtering. Specifically, users input keywords as their profile filters to indicate their personal interests, and register the filters in order to receive content of interest. When fresh content items are published, these items are matched with the pre-registered filters. By representing both content and filters as a set of terms, a boolean model or vector space model (VSM) [6] can check whether a content item matches a filter or not. Adopting the keyword-based approach is due to the following reasons. First, keywords have become a de-facto standard for users to find content of interest. Second, using keywords offers a fine-grained content filtering mechanism, and overcomes the current coarse content filtering issue, and finally many real applications have adopted the keyword-based approach, such as Google Alerts, RSS aggregator, etc. However, the alert services are still based on batch processing through search engines [13], and cannot offer timely dissemination services.

We name the keyword-based content filtering and dissemination approach by Move, and develop a system on a cluster of commodity machines for high scalability. The key of Move is to offer a scalable match algorithm. One might treat the content match as a special kind of content search. For example, search algorithms store and index content, and then answer incoming queries. For Move, the registered filters and published content might be treated as the stored documents and incoming queries in the search algorithms, respectively.

In terms of scalable search algorithm on large clusters, we use the *distributed rendezvous algorithm* (or flooding) [5] for illustration, since it is probably one of the most successful algorithms for scalable search. In order to balance the storage over a cluster of machines, this algorithm randomly distributes documents over the machines, and indexes locally stored documents by inverted lists. Next, for scalable search, queries are sent in parallel (or flooded) to the machines. This maximizes the parallel ability for queries.

Though with a great success in content search, the distributed rendezvous algorithm cannot be simply adapted for scalable content matching, due to the *blind flooding*. That is, documents are randomly stored in the machines (for balanced storage cost). As a result, even if a machine does not contain needed content, the machine still receives queries for content search. Thus, if the blind flooding is adapted for Move, the machines have to match every received content with pre-registered filters (with the help of inverted lists of such filters). Since content articles typically contain on average tens and even thousands of terms, the match algorithm needs to retrieve inverted lists associated with all such terms, incurring high latency and low throughput (the throughput is measured by the total number of matched documents per second).

In this paper, instead of the rendezvous approach, we propose to index registered filters by a *distributed inverted list* on $O(1)$ hop DHT platforms (such as Dynamo [9] and Apache Cassandra [1]). Based on the distributed inverted list, all filters containing a term are registered on the node responsible for the term. Then, by the DHT routing, content documents are forwarded to the nodes maintaining the terms commonly appearing in such documents and registered filters. Thus, the forwarding ensures that the content and filters on destination nodes must contain common terms, and avoids blind flooding.

Next, to balance the number of processed documents and number of registered filters on each node, we propose to allocate (i.e., replicate and separate) filters for high throughput. That is, (i) if a node receives too many documents, the filters on the node are then replicated to multiple nodes. Next, the documents are randomly forwarded to each of such nodes, and such nodes receive a balanced number of documents. The replication hopefully balances the number of processed documents, but leading to more replicated filters. This involves a tradeoff between processed documents and registered filters. (ii) If too many filters are registered on a node, more nodes are then used to cooperatively store the filters for balanced storage cost. Next a document is in parallel forwarded to such nodes, maximizing the parallel forwarding ability.

Finally, we study an optimization problem to maximize the throughput of matching documents with filters, meanwhile keeping the associated overhead (i.e., the overhead for storing filters) as a constant constraint (due to the capability limit). To solve the problem, we derive adaptive factors to allocate filters, including how to allocate filters, and how many nodes are assigned to place allocated filters.

As a summary, our contributions are given as follows.

(i) A novel filter allocation approach to increase the throughput, by the tradeoff between the number of processed documents and number of registered filters.

(ii) An adaptive approach to assign the number of nodes for allocated filters.

(iii) An extensive evaluation with real data sets over a cluster of approximately 100 commodity machines to verify that the proposed solution can achieve higher throughput than the literature approaches.

The rest of this paper is organized as follows. Section II investigates related work. Section III introduces a baseline solution, and discuss its challenges. Next, Section IV gives an adaptive allocation approach, and Section V presents the system design. After that, Section VI evaluates the proposed solution. Finally, Section VII concludes the paper.

II. Related Work

Overview of Inverted List: An inverted list is a data structure to index documents. It stores a mapping from each term to a set of documents (precisely, pointers of documents). The set, typically implemented as a *posting list*, maintains all documents containing the term. Though the inverted list is frequently used to index documents (typically inside a machine), its distributed version does not work efficiently to index documents on large clusters. That is, the distributed inverted list maps each term to a unique machine which stores all documents containing the term. Given large articles containing at least tens and even thousands terms, in the worst case, an article might be mapped to tens and even thousands machines. This incurs significantly high storage redundancy. Thus, real applications seldom adopt the distributed inverted list for scalable search.

Note that for the distributed inverted list approach in KLEE [15], the node responsible for a term t_i maintains a pair of $\langle$document ID, weight score of the term$\rangle$, instead of pairs of all terms in the document. However, it requires more rounds of communication to compute complete relevance scores (e.g., VSM model) and leads to high latency and low throughput.

Document filtering and dissemination: Generally our work belongs to the area of information retrieval (IR)-based filtering and dissemination. As pioneering work, SIFT [25] used keywords as profile filters, and performed centralized information filtering (where all filters were locally available). The previous work STAIRS [17], [21] studied in the problem to reduce the forwarding cost of matching documents in P2P networks, and the core scheme is the term selection algorithm based on a filter indexing structure. Instead, this paper focuses on the Move optimization problem for high throughput, and target on large clusters consisting of commodity machines.

In content-based publish/subscribe [7], [10], content consists of data tuples and filters are typically predicates of filtering conditions like $=, <, >$, etc. However, such data models significantly differ from our focused IR context. In addition, the previous works [18], [19] proposed proactive caching techniques to speedup the routing performance of DHT networks. Sharing some similar high-level idea, this paper focuses on $O(1)$-based clustered platforms.

Scalable search and dissemination on large clusters: Real web search engines, e.g., Google [5], use the distributed rendezvous (Section I gave an overview of this approach) for scalable search on large clusters. [16] improved [5] by studying the problem of changing partitions of a running system and proposed to on-the-fly re-configure the partition level. However, [16] is still a distributed rendezvous algorithm, and is inefficient for content filtering. The experiment shows that the proposed solution outperforms [16].

Similar to our work, StreamCloud [11] provided the timely processing of continuous data flows for sharing-nothing clusters. The main focus of StreamCloud is the strategy of query parallelization with a very high volume of stream tuples, but with several queries. Instead, our problem considers a very large number of documents and filters, and the main concern is the latency of matching documents with filters.

Key/value platforms Amazon's Dynamo [9] provided highly available and scalable key/value platform with $O(1)$ DHT routing. With the help of Gossip protocol, every node in Dynamo maintains information about all other nodes. Besides Amazon, key/value platforms are widely used in large scale online services, such as Facebook, Digg, etc. The open source project Apache Cassandra [1] supports a BigTable data model [8] running on an Amazon Dynamo-like infrastructure. We implement our prototype on Cassandra.

The key/val platform maps a key to a unique node, and the node then stores the value associated with the key. For convenience, we call such a node as the *home node of the key*. Based on the key/value platform, the `put` function is used to store the object, and the `get` function to lookup an object associated with an input key.

III. A Baseline Solution

In this section, we first introduce the data model of the keyword-based matching problem, then give a baseline solution, and finally discuss the associated issues.

A. Data Model

There exist various types of content: textual documents, annotated binary content, media, etc. We mainly focus on documents (including web documents, blogs, RSS feeds, etc.). Nevertheless, our techniques can be adapted to other content.

We assume that a fresh document d is preprocessed, and represented by a set of $|d|$ terms t_i, $1 \leq i \leq |d|$. For the sake of convenience, we slightly abuse the notation and refer both to the document and its associated term set by d. Other content types (e.g., binary or media) typically include associated tag keywords and descriptions. Such tags can be viewed as the equivalent of terms for the purpose of content search.

Users register filters to indicate their personal interests, and expect to receive matched documents. A filter f is represented by a set of $|f|$ terms $\{t_1, .., t_{|f|}\}$. Similar to d, the notation of f refers both to the filter and its associated term set.

Given a document d and a filter f, we say that d successfully *matches* f (or f matches d), if there is a term t that appears inside both d and f. In order words, we assume a *boolean*-based matching approach. However, our solution can be extended to approaches with more involved matching semantics, such as similarity thresholds-based semantics [25], [17]. Following [25], [17], we can extend our solution to support such semantics.

B. Baseline Solution

Based on the data model above, we build a baseline solution on key/value platforms to *register filters* and *disseminate documents*. First, to register a filter f, by the `put` function, the full information of f is locally stored on the home nodes of all query terms in f. Then, the home node of t_i registers all filters containing t_i. For all filters f registered on the home node of t_i, we use an inverted list to index such filters f. Now the key point is *the inverted list on the home node of t_i builds only the posting list associated with t_i*. That is, though the filters f contain a term t_j ($\neq t_i$), the home node of t_i will not build the posting list for such t_j (because the home node of t_j can build the posting list for t_j). Intuitively, the posting lists on all home nodes form a distributed inverted list. For convenience, we name this approach as a *distributed inverted list* approach. In Figure 1, the filter $f_1 = \{A, E\}$ is stored in the two nodes n_1 and n_5, which are the home nodes of terms A and E, respectively. The node n_1 registers all filters containing the term A, i.e., the 5 filters $f_1 ... f_5$, and builds only one posting list for A. The posting lists on all home nodes ($n_1 ... n_5$) form a distributed inverted list to index all registered filters.

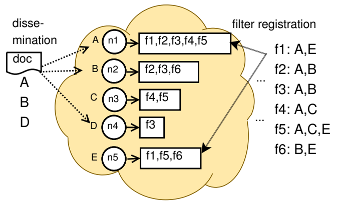


Fig. 1. Baseline Solution

Note that the above registration approach will not incur high storage redundancy because a filter contains on average 2-3 query terms. However, if used for indexing documents, it does incur high redundancy, because of the large number of terms appearing in documents. Thus, almost all real applications do not adopt the distributed inverted list approach to index documents on large clusters.

After filters are registered, the dissemination is as follows. When a document d is published, by the `get` function, d is *in parallel* forwarded to the home nodes for the $|d|$ terms $t_i \in d$. The home nodes of t_i then match d with registered filters and disseminate d to those matching filters. Again, as the key idea, on the home node of t_i, *the centralized matching algorithm retrieves only the posting list associated with t_i*, even if there exist posting lists for t_j ($\neq t_i$). For example in Figure 1, the document d contains three terms A, B and D. Next, the document d is in parallel forwarded to the three home nodes for terms A, B and D. On the home node of A, only the posting list of A is retrieved to find the referred filters, i.e., $f_1 ... f_5$. Clearly, this centralized matching algorithm greatly reduces the latency of matching d with locally registered filters.

C. Issues

The baseline solution does work and can correctly find all matching filters. However, it suffers from the issue of *low throughput* that is shown as follows.

The throughput is measured by the average number of documents d that are disseminated to users per period. For the document dissemination, the key is the matching algorithm to find the filters matching d. Thus, for each node to process the document matching, the throughput of the node can be alternatively measured by the average latency to find the filters matching d. Lower latency indicates higher throughput.

We notice that the low throughput of the baseline solution is incurred by the skewed distribution of both document terms and query terms (Section VI will show the skewed distribution). First, some terms t_i frequently appear in a very large number of documents (such documents are denoted by $\mathcal{Q}_i$). We define the *frequency* q_i as the rate between $|\mathcal{Q}_i|$ and the total number Q of all documents in the period (i.e., $|\mathcal{Q}_i| = q_i \cdot Q$). Then, the home nodes of t_i receive $q_i \cdot Q$ documents. If q_i is high, the home nodes suffer from a hot spot issue.

Next, some terms t_i appear in a very large number of filters, leading to unbalanced storage. Suppose $\mathcal{P}_i$ is the set of all filters containing t_i. We then define the *popularity* p_i as the rate between $|\mathcal{P}_i|$ and the total number P of all filters (i.e., $|\mathcal{P}_i| = p_i \cdot P$). Due to the skewed p_i, some terms t_i are very popular, and the home nodes of t_i have to store $p_i \cdot P$ filters containing t_i. Given a very large $p_i \cdot P$, the local disk unfortunately becomes the performance bottleneck, incurring high matching latency.

Beyond the above intuition, we further prove that the MPAF optimization problem, i.e., *m*aximizing the through*p*ut in order to find *a*ll *f*ilters matching d, is NP-hard, unless P=NP (The details refer to [20]).

IV. Optimal Filter Allocation

Recall that the above MPAF maximizes the throughput, but it does not consider the associated overhead. In this section, we generalize it to the Move optimization problem, i.e., maximizing the throughput while keeping the overhead as a constraint. Based on the optimal numeric results solved by Lagrange multiplier, we then approximate the results by a rounding solution.

To be consistent with the Move problem, we name our solution as the Move scheme, which contains two aspects: (i) the organization (or allocation) of filters on machines in this section, and (ii) the forwarding of documents to match registered filters in next Section V.

A. Overview

Recall that the skewed distribution of term popularity p_i and frequency q_i makes the home node of t_i store too many filters and receive too many documents. Both lead to low throughput. In particular, since content documents typically contain on average tens and even thousands of terms, matching such documents with filters incurs high latency. To achieve high throughput, our basic idea is to balance the number of processed documents and registered filters. Specifically, by storing reasonably more redundant filters (due to the capacity constraint, arbitrary high storage is impractical), we expect to reduce the number of documents to be processed. The tradeoff offers chance to achieve low latency by matching a balanced number of documents with filters, leading to high throughput. To this end, we propose to *replicate* and *separate* filters across the machines of a cluster. In detail,

Replication of filters. First, we consider that a term t_i frequently appears in a large number of documents (such documents are those in $\mathcal{Q}_i$), and the home node of t_i suffers from a hot spot issue. In order to avoid this issue, we propose to *replicate* the filters $\mathcal{P}_i$ on n_i (≥ 1) nodes. Next, the document $d \in \mathcal{Q}_i$ is randomly sent to one of the n_i nodes, and each of the n_i nodes on average receives $q_i \cdot Q/n_i$ documents. Clearly, the replication of filters can avoid the hot spot issue, and help increase the throughput of matching the $q_i \cdot Q$ documents.

Separation of filters: Second, we consider that a term t_i is very popular with a large p_i. If all the filters containing t_i, i.e., $\mathcal{P}_i$, are stored on the home node of t_i, the latency of matching a document d with $\mathcal{P}_i$ is high. To reduce the latency, we propose to let n_i (≥ 1) nodes cooperatively store the filters $\mathcal{P}_i$. That is, we *separate* the filters $\mathcal{P}_i$ to n_i subsets, each of which contains $p_i \cdot P/n_i$ filters. Next, each of the n_i nodes stores one of the n_i separated subsets. When a document d comes, it is in parallel forwarded to all such n_i nodes. Since each of the n_i nodes contains a smaller number of filters, the latency of the node to match d becomes smaller.

Given the coexistence of skewed p_i and q_i, neither the replication nor separation scheme alone can minimize the latency. For example, the separation alone does not reduce the number of received documents, and such n_i nodes still suffer from the hot spot issue, and the replication scheme still registers $p_i \cdot P$ filters on each of the n_i node. Thus, we propose the *allocation* scheme to combine both together, and balance the number of processed documents and registered filters.

In the following sections, we first consider how to allocate the filters $\mathcal{P}_i$ in terms of a specific term t_i (Section IV-B), and then by considering all terms appearing in filters, we derive optimal allocation factors for all terms (Section IV-C). The allocation factor is related to n_i, the number of nodes that are assigned to allocate the filters $\mathcal{P}_i$, such that when all terms are considered, the overall throughput is maximized.

B. Allocation of Filters $\mathcal{P}_i$

For a specific term t_i, the allocation of filters $\mathcal{P}_i$ indicates (i) how many subsets the filters $\mathcal{P}_i$ are separated to and (ii) how many copies each filter $f \in \mathcal{P}_i$ is created. To bound the overhead (i.e., the storage cost) of allocating filters $\mathcal{P}_i$, we assume that the filters in $\mathcal{P}_i$ are allocated to n_i nodes. The number n_i will be computed in Section IV-C.

To find minimal latency, we define an *allocation ratio* $r_i \in [1/n_i, 1.0]$ to allocate the number of the copies for the filters $\mathcal{P}_i$ among the n_i nodes. In detail, the n_i nodes are divided into $1/r_i$ partitions and each partition contains $r_i \cdot n_i$ nodes. Then, the $p_i P$ filters are mapped (i.e., allocated) onto such partitions. Specifically, the filters are separated into $r_i \cdot n_i$ subsets, and each subset contains $p_i \cdot P/(r_i \cdot n_i)$ filters. Meanwhile, the subset is replicated onto the $1/r_i$ nodes with $1/r_i$ copies.

When a document $d \in \mathcal{Q}_i$ comes, one of the $1/r_i$ partitions is selected. Then d is in parallel forwarded to all $r_i \cdot n_i$ nodes in the selected partition. Given the $1/r_i$ partitions and $q_i \cdot Q$ documents, each node in a partition on average receives $q_i \cdot Q/(1/r_i) = r_i \cdot q_i \cdot Q$ documents. As a summary, every node stores $\frac{p_i \cdot P}{n_i \cdot r_i}$ filters, and receives $(q_i \cdot Q \cdot r_i)$ documents.

Figure 2 shows an example of the allocation for $n_i = 12$ nodes (i.e., the nodes $n_1 ... n_{12}$), $\mathcal{P}_i = 8$ filters ($f_1 ... f_8$) and $\mathcal{Q}_i = 3$ documents ($d_1 ... d_3$) and $r_i = 1/3$. The 12 nodes are divided to 3 partitions (i.e., 3 rows) and each partition contains 4 nodes (i.e., 4 columns). Next, the 8 filters are separated to 4 subsets

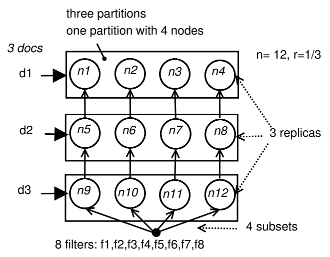


Fig. 2. Filter Allocation

and each subset contains 2 filters. Each filter is replicated on 3 nodes, leading to 3 copies. For example, the two filters f_1 and f_2 inside the same subset are replicated to the three nodes of n_1, n_5 and n_9. When 3 documents come, each of these 3 documents is sent to one randomly selected partition for document matching. Then, the document (e.g., d_1) is in parallel forwarded to all nodes inside the partition (e.g., the 4 nodes $n_1 ... n_4$ in the first partition). As a summary, each node receives 1 document and stores 2 filters.

Clearly, when the allocation ratio $r_i \in [1/n_i, 1.0]$ is smaller, the nodes are divided with more partitions, and the filters are replicated with more copies. Here, $r_i = 1/n_i$ is the special case with n_i partitions and one node per partition, and each filter is replicated onto n_i nodes with n_i copies (i.e., the pure replication approach in Section IV-A), and $r_i = 1$ is the special case with only one partition containing n_i nodes, and no filter is replicated (i.e., the pure separation scheme in Section IV-A).

Based on the above ratio r_i, we compute the latency Y_i of matching $q_i \cdot Q$ documents with $p_i \cdot P$ filters as follows.

1) A simple approach to compute Y_i: The precise latency Y_i involves many parameters, e.g., the network bandwidth in the cluster, the local disk, CPU, etc. We first consider a simple approach to compute Y_i as follows. We assume that Y_i is mainly decided by the latency of matching documents with the filters that are locally stored in the disk, and ignore the network latency of transferring documents to the n_i nodes. This assumption is consistent with the recent measurement work [24]. The extensive measurement results on Amazon EC2 platform show that the disk IO is the main bottleneck of the cloud platform. For our case, on each node, filters are indexed by a local inverted list, and the latency of the node to match a document d with filters mainly depends upon the time to retrieve posting lists from the local disk.

Next, based on the proposed allocation approach, the filters $\mathcal{P}_i$ are now stored on a set of nodes (i.e., the nodes in the partitions). As shown in Section III-C, due to the parallel forwarding, the latency Y_i of matching the documents $\mathcal{P}_i$ with the filters $\mathcal{P}_i$ mainly depends on the latency of each node in the partitions to match documents with locally stored filters. Following the above assumption, we compute Y_i as follows.

Suppose that y^p is the average latency of matching a document d with a filter of $\mathcal{P}_i$. Then, Y_i depends on the numbers of documents and filters to be matched. Recall that for each node inside one divided partition, it locally stores $\frac{p_i \cdot P}{n_i \cdot r_i}$ filters, and receives $(q_i \cdot Q \cdot r_i)$ documents. Therefore, the latency Y_i is computed as follows.

$$Y_i = y^p \cdot (q_i \cdot Q \cdot r_i) \cdot \frac{p_i \cdot P}{n_i \cdot r_i} = y^p \cdot \frac{p_i \cdot P \cdot q_i \cdot Q}{n_i} \qquad (1)$$

In the equation above, $Y_i = y^p \cdot \frac{p_i \cdot P \cdot q_i \cdot Q}{n_i}$ is independent upon r_i. Though such a result is based on the aforementioned assumptions, it gives the following insightful results (for simplicity, we assume that y^p is a constant average).

- For a term t_i, the latency Y_i varies with its popularity p_i and the frequency q_i (suppose P and Q are given). Therefore, a larger $p_i \cdot q_i$ indicates higher latency.
- The latency Y_i also depends upon n_i. If more nodes are assigned for t_i, i.e., a larger n_i, then Y_i is reduced.

2) An extended approach to compute Y_i: Now, let us consider the latency of both transferring documents and matching documents as follows.

$$Y_i = (q_i \cdot Q \cdot r_i) \cdot (y^d + y^p \cdot \frac{p_i \cdot P}{n_i \cdot r_i}) = (q_i \cdot Q) \cdot (y^d \cdot r_i + y^p \cdot \frac{p_i \cdot P}{n_i}) \quad (2)$$

In the above equation, y^d is the average latency to transfer a *d*ocument d to a node inside a partition, and y^p is the average latency to match d with one locally stored filter.

On the condition that each of the n_i nodes provides enough capacity (denoted by C) to store $\frac{p_i \cdot P}{n_i \cdot r_i}$ filters, a smaller r_i, e.g., $r_i = 1/n_i$, leads to lower latency Y_i. This makes sense: when each node has an enough capacity to store $\frac{p_i \cdot P}{n_i \cdot r_i} = p_i \cdot P$ filters (due to $r_i = 1/n_i$), a smaller r_i indicates that more nodes are assigned and the $q_i \cdot Q$ documents are forwarded in a more parallel manner, leading to high throughput.

Consider the case that each of the n_i nodes cannot provide enough capacity C to store $p_i \cdot P$ filters, i.e., $C < p_i \cdot P$. We then tune r_i from $1/n_i$ to a larger value r'_i, until $C \geq \frac{p_i \cdot P}{n_i \cdot r'_i}$ holds. We define $\alpha_i = r'_i / r_i = r'_i \cdot n_i$ as a *tuning ratio*. When n_i and P are given, a larger p_i leads to a linearly increased α_i. Section IV-C will use it to achieve an optimal result of n_i.

Equations 1 and 2 benefit from a smaller r_i. Considering that $r_i \geq 1/n_i$, we find that a large n_i helps achieve a smaller r_i. However, a larger n_i incurs more storage overhead, because the filters $\mathcal{P}_i$ are replicated with $1/r_i$ copies. To overcome this issue, the following section avoids assign a very large n_i for the term t_i, and then minimizes the overall latency.

C. Computing Filter Allocation Factor

Section IV-B computes the latency Y_i for a specific term t_i. Now, for the total T terms that appear in all filters, the Move scheme further computes the overall latency Y of matching all Q documents with P filters. We consider an optimization problem to minimize Y by setting an optimal n_i for each term t_i. We first use Equation 1 to compute the overall latency Y and then consider an extension based on Equation 2.

Recall that for a specific term t_i, Equation 1 computes the latency Y_i of matching the documents Q_i with the filters $\mathcal{P}_i$. When there exist T query terms, the overall average latency Y is computed by $Y = \frac{1}{T} \cdot \sum_{i=1}^{T} Y_i = \frac{1}{T} \cdot \sum_{i=1}^{T} (\frac{p_i \cdot P \cdot q_i \cdot Q}{n_i})$.

Besides the latency Y, we need to consider the constraint $\sum_{i=1}^{T} (n_i \cdot p_i \cdot P) = N \cdot C$. In this constraint, $\sum_{i=1}^{T} (n_i \cdot p_i \cdot P)$ is the largest storage overhead, incurred by replicating the filters

$\mathcal{P}_i$ with at most n_i copies (due to $r_i \geq 1/n_i$); and $N \cdot C$ is the overall capacity provided by N nodes, each of which offers the storage capacity of C.

Now, we have the following result with respect to n_i.

Theorem 1 *Given the constraint $\sum_{i=1}^{T} (n_i \cdot p_i \cdot P) = N \cdot C$, the overall latency $Y = \frac{1}{T} \sum_{i=1}^{T} (\frac{p_i \cdot P \cdot q_i \cdot Q}{n_i})$ is minimized if $n_i \propto \sqrt{q_i}$.*

Theorem 1 (We use Lagrange multiplier to solve the optimization problem in Theorem 1. Due to the space limit, the proof refers to the technical report [20]) derives the optimal numeric result n_i is proportional to the square root of q_i, i.e., $n_i \propto \sqrt{q_i}$. By classic rounding solutions, e.g., randomized rounding [12], we approximate an integer n_i.

Note that the result $n_i \propto \sqrt{q_i}$, independent upon p_i, is due to the assumption that $r_i = 1/n_i$ and the storage capacity C of each node is enough large than $p_i \cdot P$. Considering that the general case $r_i \geq 1/n_i$, we follow the tuning process in Section IV-B2 to find the tuning ratio $\alpha_i = r'_i \cdot n_i$. Given α_i, which linearly depends on p_i, we then similarly achieve an optimal result $n_i \propto \sqrt{q_i \cdot p_i}$. Next, based on the p_i and q_i for each term t_i, with the system-level parameters N, C etc., we then compute a specific value n_i for each t_i.

Next, based on Equation 2, we consider the extension of Theorem 1. Similar to Theorem 1, we first assume that $r_i = 1/n_i$ and $C > p_i \cdot P$ hold. Thus, if we substitute $r_i = 1/n_i$ back to Equation 2 and get

$$Y_i = \frac{q_i \cdot Q \cdot (y^d + y^p \cdot p_i \cdot P)}{n_i} \tag{3}$$

Following Theorem 1, we have the following result:

Theorem 2 *Given the constraint $\sum_{i=1}^{T} (n_i \cdot p_i \cdot P) = N \cdot C$, the overall latency $Y = \sum_{i=1}^{T} \frac{q_i \cdot Q \cdot (y^d + y^p \cdot p_i \cdot P)}{n_i}$ is minimized when $n_i \propto \sqrt{(1 + \beta \cdot q_i)}$, where $\beta = y^p \cdot P / y^d$.*

Theorem 2 (due to the similar proof with Theorem 1, we skip the proof of Theorem 2) has the similar result as Theorem 1, except the subitem $\beta = y^p \cdot P / y^d$. The subitem β indicates the ratio between the latency to match a document with P filters and the latency of transferring a document to a node. When the number of P is very large, we can reasonably arrive at $\beta \gg 1$, and Theorem 2 is then consistent with Theorem 1. Also in the general case that the storage capacity C of a node is smaller than $p_i \cdot P$, we similarly have the result $n_i \propto \sqrt{p_i \cdot q_i}$.

V. System Design

Overview: We design a system prototype over the Apache Cassandra 0.87 platform [1], an open source implementation of Dynamo [9]. The system mainly contains three data stores and a forwarding engine, which are shown in Figure 3.

On each node, the *data stores* (using the Big Table scheme [8] supported by Apache Cassandra) are filter stores for registered filters, local inverted list to index the stored filters, and meta data store that is used to maintain the statistics of coming documents and registered filters (such as p_i and q_i).

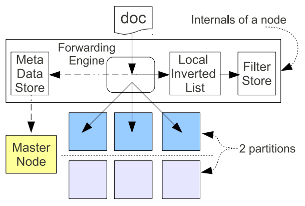


Fig. 3. Internals of the developed system

The *forwarding engine* uses a main memory-based forwarding table to maintain the addresses of the nodes that store allocated filters. As a simple approach, on a node, say the home node of t_i (the node is denoted by m_i), the forwarding table maps t_i to a two-dimensional array consisting of $1/r_i$ rows (i.e., partitions) and $n_i \cdot r_i$ columns (Section IV gives n_i and r_i). The elements of the array refer to those nodes which store allocated filters. All filters in $\mathcal{P}_i$ are stored on the home node t_i and the nodes referred by the elements of the forwarding table.

Note that the total number of terms is very large, for example, millions of terms. Given a cluster with hundreds or at most thousands of nodes, the number T_i of terms maintained by the node m_i is correspondingly large. Following the above simple solution, m_i has to maintain T_i two-dimensional arrays in the forwarding table. Due to the existence of millions of terms, T_i is large and the associated maintenance cost is nontrivial.

To overcome the above issue, for all terms t_i maintained on the node m_i, we sum the associated p_i and q_i to represent the *node popularity* p'_i and the *node frequency* q'_i with respect to the node m_i. Then, we treat all filters registered on m_i as a single set of filters $\mathcal{P}'_i$. Next, Theorems 2 and 3 use the p'_i and q'_i to compute a n'_i, which is used to allocate the filters in $\mathcal{P}'_i$. Now, the forwarding table on the node m_i maintains only one two-dimensional array (instead of T_i arrays) by mapping all T_i terms to the array. Clearly, the approach greatly reduces the maintenance cost of the forwarding table.

Document Dissemination When a document d comes, we can simply forward d to the home nodes of all terms $t_i \in d$ and $t_i \in BF$, where BF is the bloom filter summarizing all terms in registered filters. The term membership check helps reduce the forwarding cost. Note that, the previous work [17], [21] can help select a smaller number of terms t_i, but leading to high latency. Thus, for high throughput, we discard the selection algorithm [17], [21]. Next when d arrives at the home nodes, following the centralized matching algorithm in Section III-B, d is matched with only those filters in $\mathcal{P}_i$. Meanwhile, if the filters in $\mathcal{P}_i$ have been allocated, the forwarding engine randomly selects a partition (i.e., a row of the forwarding table), and forwards d in parallel to all nodes of the selected partition. When arriving at a node of the partition, d is matched with locally stored filters. Until now, we can ensure all matching filters, either on the home node of t_i or the nodes in such a partition, are found.

Solving the Move *optimization problem* We use a dedicate node in the cluster collects the statistics such as the node popularity p'_i and node frequency q'_i from all nodes m_i to compute the result n'_i for m_i. The dedicate node is similar to the master node in Hadoop, and harnessing redundant servers in groups can enhance the resilience to node failure. Following the discussion in Section IV, we use the general result $n_i \propto \sqrt{p_i \cdot q_i}$.

Selection of allocated nodes: To select the nodes to place allocated filters, theres are two basic options: the ring-based successors, and rank-aware nodes (both are supported by the Cassandra). For the first option, the allocated filters of $\mathcal{P}_i$ is placed to the successors of the home node of t_i along the ring of Cassandra. The second option places the allocated filters to the nodes inside the same rack as the home node of t_i. However, either of the two options has the downside. For example, since the allocation causes the movement of filters across the cluster, the successor-based option might cause network traffic. Next, the failure of a whole rack might lose stored filters.Thus, to avoid such downsides, we choose one half of the n_i nodes based on the successors, and another half based on the rack-aware nodes.

Allocation Policy: Finally, the allocation policy involves when to allocate filters. One option is the *passive allocation*, by which the allocation occurs after the patterns of filters and documents (i.e., the distributed of p_i and q_i) are learned. However, the passive allocation suffers from the issue: at the moment that the term t_i is associated with the large p_i and q_i, the home node of t_i suffers from the issues of hot spot and heavy workload to match documents. The passive allocation needs to move the filters $\mathcal{P}_i$ across the cluster, and further aggravates the workload of the home node. To overcome the issue, another policy is the *proactive allocation* as follows. Filters represent the personal interests of end users, and end users do not change the filters very frequently [14]. Since the filters are registered before document publication, it is easy to learn the pattern of filters (the distribution of p_i). In addition, for q_i, an offline approach based on the existing document Corpus could achieve a relatively high precision [22]. Based on such p_i and q_i, we approximate the allocation factor n_i. Next, when the pattern of filters and documents is available, we refine the allocation factor.

VI. Evaluation

A. Experimental Settings

We first report the used datasets, and then present the experimental methodologies (including performance metrics).

(1) Profile Filters: There is no publicly available data trace with respect to the keyword-based profile filters (such as inputs of Google Alerts). We opt to use the data traces of traditional web search services, which are truly representative of the end user behavior to use keywords in the real world. Similar approaches are used by many previous works [23]

We use an input query history file (containing 4,000,000 pre-processed queries) collected from the Microsoft *MSN* search engine (*MSN* in short). On average, the number of terms per query is 2.843 in *MSN*. Furthermore, the cumulative percentage of all filters containing at most 1, 2 and 3 terms is 31.33%, 67.75%, and 85.31%, respectively. These numbers indicate that real users prefer to use short queries, on average only 2–3 terms. Thus, Move registers filters by the distributed inverted list approach, without incurring too much storage redundancy. Also replicating filters benefits from the short queries and will not lead to high storage cost.

For the popularity p_i of query terms, we plot the ranked p_i of the *MSN* trace in Figure 4, where x-axis is the ranking Id of a term t_i, and y-axis is the associated p_i. This figure clearly indicates the skewed distribution of p_i. For example, among the total 757996 distinct query terms, the accumulated popularity value of the top-1000 terms is 0.437, indicating such top-1000 terms are very popular.

(2) Content Documents: We use two datasets. The first one is the Text Retrieval Conference (TREC) WT10G web corpus [3], which is a large test set commonly used in web information retrieval. The dataset contains around 1.69 million Web page documents (a total of 10 Gigabytes). The average size of each document is 5.91KB, and the average number of terms per document is 64.8. The data set was pre-processed with the Porter algorithm [3] and common stop words such as “the”, “and”, etc. were removed from the data set.

The second dataset is based on TREC AP, a text categorization task based on the Associated Press articles used in the NIST TREC evaluations [3]. Compared with the TREC WT10G dataset, the TRACE AP set is composed of fewer (1,050) articles but with a larger number (on average 6054.9) of terms per article.

In Figure 5, we plot the ranked frequency of the document terms in two TREC datasets (due to the very large number of documents in both traces, we only plot the top-10^5 frequency rates). This figure similarly indicates the skewed distribution of the frequency rates in both traces. By computing the entropy values of the frequency rates: 9.4473 for the TREC AP and 6.7593 for TREC WT, we easily verify that the frequency rates of the TREC WT is skewer than the TREC AP.

In addition, we are interested in whether or not those popular query terms also frequently appear in the documents. Among the top-1000 popular query terms, 26.9% of them are among the top-1000 frequent document terms in the TREC AP dataset, and 31.3% of them are among the top-1000 frequent document terms in the TREC WT dataset. Thus, for such terms, it is necessary for the Move solution to combine both replication and separation schemes, and neither the replication nor separation scheme alone can achieve high throughput.

(3) Experimental Methodologies: We first register all filters, and then use multiple clients to injects documents. Each client injects 1000 documents per second. By using more clients, we can increase the rate of injecting documents.

We compare the Move solution with two other approaches: RS (the distributed *r*endezvou*s* scheme, based on the acquired source code from [16]), and IL (the pure distributed *i*nverted *l*ist scheme without allocating filters).

- For the RS, by extending [16], we map the hash ID of

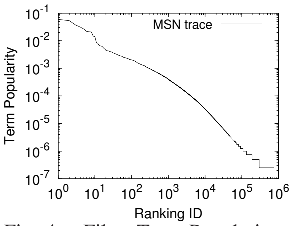


Fig. 4. Filter Term Popularity

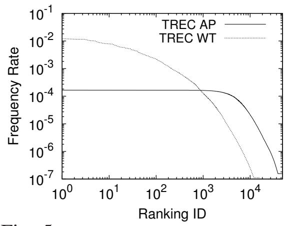


Fig. 5. Document Term Frequency

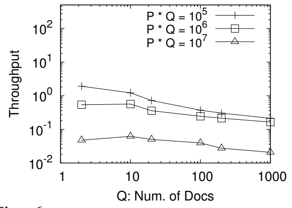


Fig. 6. A Single Node. (TREC AP)

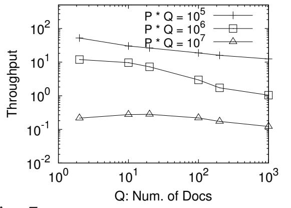


Fig. 7. A Single Node. (TREC WT)

a filter f's unique name on the associated home node, such that all filters are evenly distributed across the machines of the cluster. Meanwhile, because the RS does not support the distributed inverted list to register filters, on each node, we have to index locally registered filters by an inverted list, and then match documents d with locally registered filters by using the classic centralized algorithm SIFT [25]. That is, to match a document d with locally registered filters, with the help of the local inverted list, SIFT [25] retrieves all posting lists associated with the $|d|$ document terms, and then find the filters referred by the item of such posting lists.

- The IL is the same as the Move, except allocating filters.
- For the Move, to enable filter allocation, for each TREC data set, we use the 1000 documents as the offline document corpus to approximate q_i (and p_i can be approximated based on pre-stored filters). Next, every 10 minutes, the values of q_i are renewed based on new incoming documents. Based on the statistics of p_i and q_i, filters are then allocated periodically.

We measure the throughput of the three schemes by the number of processed documents. For a document, if all matching filters are found, we then add the throughput by 1. After all documents are published, we measure the overall average throughput per second. Besides, we study the load distribution of the nodes and the availability under node failure.

B. Experimental Results on a Single Node

Recall that the key idea of Move is to tradeoff the number of incoming documents and the number of stored filters on each node. Thus, before going to the cluster environment, we first study, on a single node how the, number Q of processed documents and the number P of registered filters affect the throughput. We fix the product value R $(= P \times Q)$ and study how P and Q affect the throughput, and which one is the major factor to decide the throughput. Using the product is based on the theoretic result $n_i \propto \sqrt{p_i \cdot q_i}$.

As shown in Figures 6 and 7, we respectively use the documents of the TREC AP and WT datasets to match the registered filters (i.e., the entries of *MSN* trace). In addition, for the scalability test, we also follow the distribution of queries terms in the *MSN* trace to synthetically generate filters, such that the number of generated filters is larger than the number $(= 4 \times 10^6)$ of entries in the *MSN* trace.

First in Figure 6, for the TREC AP, we consider three scenarios with $R = (P \times Q)$ equal to 10^7, 10^6 and 10^5, respectively. The x-axis shows the number Q of documents, and the y-axis plots the throughput to match Q documents with P $(= R/Q)$ of filters. In this figure, on the overall, a larger Q (except $Q = 10$) leads to lower throughput, or equally, due to the fixed R, a larger P leads to higher throughput. For example with $R = 10^6$, when P grows from 10^3 to 5×10^4 (i.e., Q is decreased from 200 to 10), the throughput is increased by around 8.92 folds. The reason that a larger P and smaller Q lead to higher threshold is as follows. Since documents typically much more terms than filters, reducing the number of processed documents can hopefully reduce the matching latency and has the chance to increase the throughput.

Note that a smaller Q does not certainly mean higher throughput. For example, for $R = 10^7$, the throughput of $Q = 2$ (i.e., $P = 5 \times 10^6$) is even slightly lower than $Q = 10$ (i.e., $P = 10^6$) . That is because when P is very large, the disk IO becomes the performance bottleneck and the processing time becomes larger. Thus, by severing a smaller number Q of documents and storing a reasonably larger number P of filters (i.e., limiting the number P of filers smaller than a bound C, e.g., 5×10^6 in this figure), we have chance to reduce the processing time and increase the throughput.

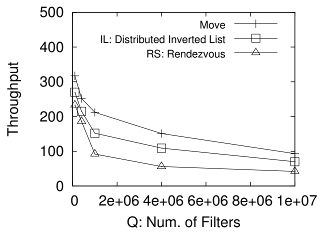
In addition, in terms of the different values of R, a larger R obviously leads to more processing time. For example, given $D = 1000$, the processing time for $R = 10^7$ is around 6.714 folds of the one for $R = 10^5$.

When comparing the two Figures 6 and 7, we find the throughput of the TREC WT is much higher than the TREC AP. For example given $R = 10^6$ and $Q = 100$, the throughput of TREC WT is around 81.84 times of TREC AP. This is roughly consistent with the fact that the average number of terms per article in TREC WT (64.8 terms per article) is much smaller than the one in TREC AP (6054.9 terms per article).

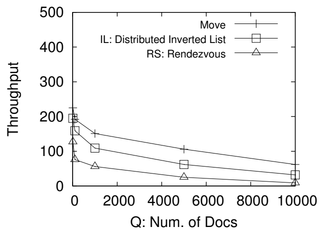
In summary, there exists chance to increase the throughput, if we could reduce the number Q of documents to be processed, and meanwhile increase a reasonably larger number P of redundant filters (e.g., caused by allocated filters). This result motivates the Move solution to design the tradeoff between the number P and the number Q.

C. Scalability on a Cluster Environment

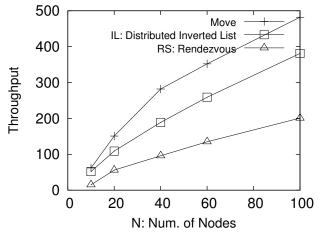
In Section VI-C, we select around 100 nodes from our department Ukko cluster [4]. The cluster has 240 Dell PoweEdge M610 nodes. All nodes have 32GB of RAM and 2 Intel Xeon E5540 2.53GHz CPUs. On the cluster, we study how the Move solution achieves higher throughput. Note that the typical key/value based systems (including Dynamo, etc.) replicate each object with three replicas. Suppose the storage cost per



(a) F: Num. of Filters



(b) d: Num. of Docs



(c) N: Num. of Nodes

Fig. 8. A Cluster with Multiple Nodes

node used by the rendezvous solution is C (containing three folds of replicas of filters). Then, in the Move solution, we assume that each node stores at most C filters.

In this experiment, on a cluster of multiple nodes, we study the performance of three schemes (Move, RS and IL) by changing the total number P of registered filters, the total number Q of injected documents per second, the number N of used nodes. In addition, we limit each node registers at most $C = 3 \times 10^6$ filters (including replicas of filters). The number P of filters, number Q of documents, and number N of nodes by default are 4×10^6, 10^3 and 20, respectively. In addition, since the total number of documents in TREC AP is fewer than 10^5, we choose TREC WT as documents.

First we vary the total number P of filters from 10^5 to 10^7, and measure the average throughput of matching the default 10^3 documents with such filters. As shown in Figure 8 (a), more filters obviously lead to lower throughput. This is because more filters lead to larger processing time per document. In addition, when compared with three schemes, for example, with $P = 10^7$, the throughput of the Move, RS, and IL is 93, 70, and 42, respectively. This figure clearly shows that the Move has the highest throughput, and the IL has the lowest throughput. We discuss the three schemes as follows.

- Due to the blind flooding of the RS, each node has to match every received document d with registered filters and retrieve $|d|$ posting lists, incurring higher latency. In particular, the partition mechanism [16] leads to more redundant filters on each node, and this further increases the average processing time of each document.
- The IL, due to the co-occurrence of skewed p_i and q_i, suffers from the lowest throughput, because the "busiest" node with the hot issue and the "slowest" node registered with the largest number of filters significantly degrade the throughput of the IL.
- The proposed Move balances the number of receiving documents (i.e., no hotspots) and the number of registered filters (i.e., balanced storage cost). Compared with RS and the used centralized matching algorithm SIFT, the Move guides content forwarding only to the home node of selected terms, and ensures that the latency of matching a documents with locally registered filters is low by retrieving only the needed posting lists.

Second, we study the effect of the number Q of injected documents. As shown in Figure 8 (b), a larger Q leads to more processing time of all three schemes. For example, when Q grows 10 to 1000, the throughput of the Move, RS and IL is decreased by 3.62 folds, 6.09 folds, and 14.11 folds, respectively. The reason that the running time of the Move does not grow significantly is as follows. Among all those nodes that store the replicas of needed filters, the Move randomly chooses one of such nodes. Thus, each node of such nodes serves a subset of incoming documents, and will not lead to significantly lower throughput.

Third, we shows how the number N of used nodes affects the throughput of three schemes. As shown in Figure 8, more nodes help achieve higher throughput for all three schemes. It is obvious because given more nodes, each node registers a smaller number of filters and receives fewer documents., Thus, all three schemes use less time and achieve higher throughput to match fewer documents with such filters.

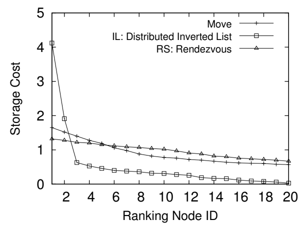
D. Maintenance

In this subsection, on the default settings of the cluster environment, we study the maintenance of the Move scheme, including load distribution, and fault tolerance to node failure.

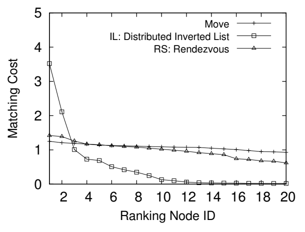
In Figures 9 (a-b), we show the load distribution on the default 20 nodes of three schemes. Instead of plotting the load of each node for three schemes, we use the rate between the load (such as stored filters or received documents) of each node and the overall average load of the RS scheme. Thus, we clearly compare three schemes.

In Figure 9 (a), the x-axis is the ranking Id of the 20 nodes, and y-axis is the associated storage cost. The IL does not allocate filters, and the skewed distribution of the term popularity p_i leads to the most skewed storage cost. Instead, the Move allocates filters across the machines of the cluster, and helps achieve balanced storage. For the RS, the consistent hash function ensures the most even storage distribution. Note that, as shown in this figure, the storage distribution of the RS is more even than the Move. This is because the allocation policy of the Move considers the distribution of both p_i and q_i, and thus does not completely balance the storage cost (i.e., an even distribution of p_i).

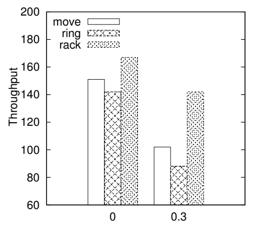
Figure 9 (b) shows the ranked matching cost of three schemes, where the matching cost is related to the number of received documents that a node needs to retrieve the local inverted list. As before, for the IL, the skewed distribution of term frequency q_i leads to the most skewed distribution of the dissemination cost. However, in this figure, the dissemination cost of the Move is more even than the RS. That is, the



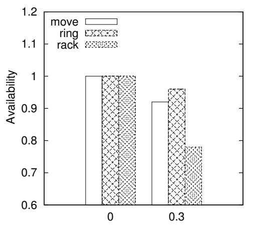
(a) Distribution of Storage cost



(b) Distribution of Matching cost



(c) Effect of Node Failure on Throughput



(d) Effect of Node Failure on Availability

Fig. 9. Maintenance cost

allocation policy of Move prefers to use a low allocation ratio r_i, and thus documents are more randomly forwarded to the $1/r_i$ partitions, leading to more balanced matching cost. For the Move, the more even distribution of matching cost than the RS is consist with the basic idea to reduce the number of processed documents while storing reasonably more filters.

Next, Figures 9 (c-d) show the effect of node failure on the throughput and filter availability. We compare the approaches to place allocated filters respectively by the Move, the ring-based and the rack-aware approaches (See Section V). The x-axis show the rate of failed nodes; the y-axis in Figures 9 (c) shows the throughput of the three approaches, and the y-axis in Figures 9 (d) plots the rate of still available filters under failure against the case without failure. In Figure 9 (c), the rack-aware approach has the highest throughput and the ring-based approach has the lowest throughput in both cases of no failure and the 0.3 rate of node failure. However, given the 0.3 rate of node failure, the rack-aware approach suffers the lowest availability. Thus, the Move combines both policies to achieve high throughput and availability.

VII. Conclusion

In this paper, we develop a scalable content filtering and dissemination system on a cluster of machines. Based on the properties of documents and filters (e.g., large document articles contain much more terms than short filters consisting of several terms), we propose an adaptive filter allocation approach to increase the throughput of matching documents. Our experimental results over real datasets verify that the classic rendezvous algorithm cannot be simply adapted for the Move problem, and the proposed scheme can achieve folds of better throughput over the rendezvous algorithm.

Acknowledgment

Weixiong Rao and Sasu Tarkoma are partially supported by Academy of Finland (Grant No. 139144), and Lei Chen's research is supported by National Grand Fundamental Research 973 Program of China under Project No. 2012CB316200, HP IRP Project 2011, Microsoft Research Asia Grant, MRA11EG05, HKUST SSRI11EG01, and NSFC No.60003074.

We also would like to thank the anonymous ICDCS 2012 reviewers for their valuable comments that helped improve this paper.

References

[1] http://cassandra.apache.org.
[2] http://highscalability.com/blog/2010/12/31/facebook-in-20-minutes-27m-photos-102 m-comments-46m-messages.html.
[3] http://trec.nist.gov/data.html.
[4] http://www.cs.helsinki.fi/en/compfac/high-performance-cluster-ukko.
[5] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.
[6] M. W. Berry, Z. Drmac, and E. R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Rev.*, 41(2):335–362, 1999.
[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.
[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, pages 205–218, 2006.
[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.
[10] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.
[11] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, and P. Valduriez. Streamcloud: A large scale data streaming system. In *ICDCS*, pages 126–137, 2010.
[12] J. M. Kleinberg and É. Tardos. Approximation algorithms for classification problems with pairwise relationships: Metric labeling and markov random fields. In *FOCS*, pages 14–23, 1999.
[13] D. Kukulenz and A. Ntoulas. Answering bounded continuous search queries in the world wide web. In *WWW*, pages 551–560, 2007.
[14] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *Internet Measurment Conference*, pages 29–34, 2005.
[15] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB*, pages 637–648, 2005.
[16] C. Raiciu, F. Huici, M. Handley, and D. S. Rosenblum. Roar: increasing the flexibility and performance of distributed search. In *SIGCOMM*, pages 291–302, 2009.
[17] W. Rao, L. Chen, and A. Fu. Stairs: Towards efficient full-text filtering and dissemination in dht environments. *The VLDB Journal*, 20:793–817, 2011.
[18] W. Rao, L. Chen, A. W.-C. Fu, and Y. Bu. Optimal proactive caching in peer-to-peer network: analysis and application. In *CIKM*, pages 663–672, 2007.
[19] W. Rao, L. Chen, A. W.-C. Fu, and G. Wang. Optimal resource placement in structured peer-to-peer networks. *IEEE Trans. Parallel Distrib. Syst.*, 21(7):1011–1026, 2010.
[20] W. Rao, L. Chen, P. Hui, and S. Tarkoma. Move: A scalable keywords-based content dissemination on a cluster of commodity machines. In *Technical Report, the computer science department, University of Helsinki*, 2011.
[21] W. Rao, R. Vitenberg, and S. Tarkoma. Towards optimal keyword-based content dissemination in dht-based p2p networks. In *Peer-to-Peer Computing*, 2011.
[22] C. Tang, Z. Xu, and M. Mahalingam. psearch: Information retrieval in structured overlays. In *HotNets-I*, 2002.
[23] C. Tryfonopoulos, M. Koubarakis, and Y. Drougas. Information filtering and query indexing for an information retrieval model. *ACM Trans. Inf. Syst.*, 27(2), 2009.
[24] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *INFOCOM*, pages 1163–1171, 2010.
[25] T. W. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Trans. Database Syst.*, 24(4):529–565, 1999.