Scalable Matching and Ranking for Network Search

Misbah Uddin, Rolf Stadler
ACCESS Linnaeus Center
KTH Royal Institute of Technology
Email: {ahmmud, stadler}@kth.se

Alexander Clemm
Cisco Systems
San Jose, California, USA
Email: alex@cisco.com

***Abstract*—Network search makes operational data available in real-time to management applications. In contrast to traditional monitoring, neither the data location nor the data format needs to be known to the invoking process, which simplifies application development, but requires an efficient search plane inside the managed system. The search plane is realized as a network of search nodes that process search queries in a distributed fashion. This paper introduces matching and ranking for network search queries. We are proposing a semantic for matching and ranking, which is configurable to support different types of management applications—from exact matching for database-style queries to loose, approximate matching, which is appropriate for exploratory purposes. We describe an echo protocol for efficient distributed query processing that supports matching and ranking. Further, we present the design of a search node, which maintains a real-time database of operational information and allows for parallel processing of search queries. A prototype implementation on a cloud testbed shows that the network search system, on a 9-node cluster with 24 core servers, executes 200 global search queries/sec with the 75th percetile latency below 100 milliseconds and with a CPU utilization below 5%. The performance measurements, together with our design, suggest that a system of 100,000 servers processing the same load would exhibit the same overhead per server and a query latency of below 1 sec.**

Keywords*—Network search, matching and ranking, distributed query processing, distributed management, in-network management.

I. Introduction

Network search is a paradigm that stresses an information-centric view of network management [1], [2]. Its main elements are real-time access to operational and configuration data, a weakly structured data model, and a scalable search function that executes inside the managed system. Network search is specially suited for large-scale dynamic networks and networked systems, and it enables novel management functionality, such as "network googling" for root cause analysis of faults, dynamic asset tracking, and real-time network analytics.

The key difference between network search and traditional web search is that the former relates to search for real-time information inside a networked system, while the latter centers around search for non real-time data in an offline index database. While many concepts of web search are applicable to network search, the challenge is to realize them under the stringent requirements of network search.

In earlier work, we proposed a design of a network search system, including a protocol for scalable query processing. In this paper, we investigate the semantics of network search queries with respect to query matching and result ranking. Elements of our proposed solution are based on results from information retrieval and web search; they make use of the extended boolean retrieval model [3], as well as connectivity metrics and attribute frequency of information objects.

We believe that a network search system should offer, for the same query language, a range of matching and ranking semantics, in order to support different types of management applications. For some applications, database-style exact matching is appropriate, where each data item either fully matches a query or does not match at all; for others, loose, approximate matching is more suitable, where data items match a query to different degrees. The same applies to ranking: for some applications, objects with more attributes of a given type are more relevant—and thus ranked higher—than objects with fewer such attributes; for other applications, objects with recently updated attributes are more relevant than those with more stale information, etc.

Consider a search query that includes a sequence of IP addresses. For an asset tracking application, for instance, matching is best performed in such a way that exactly the objects with those IP addresses are retrieved, and their ranking order is irrelevant to the application. In contrast, consider a network-security administrator who receives a set of IP addresses from an intrusion detection system and runs a network search query with these addresses. In this case, the query semantics is such that it matches objects representing devices, flows, applications, etc., that are associated with these addresses. In the query result, highly connected devices rank high, as well as current flows.

With this paper, we make the following contributions. We introduce a semantic for matching and ranking that is tailored to network search. We show how the semantic can be realized in a distributed query processing protocol, which has sound scaling properties. Both

ISBN 978-3-901882-53-1, 9th CNSM and Workshops ©2013 IFIP

the matching and ranking functions are configurable. The possible configurations range from supporting exact matching, which allows database-style retrieval, to loose, approximate matching, which is similar to the matching semantics of the vector space model in information retrieval [4], and which supports exploration of the information space. Configuration parameters can be given at query invocation time, and our design allows a network search system to concurrently process queries with different matching and ranking semantics. The paper further presents the design of a search node supporting multicore hardware. Lastly, we report performance figures from a prototype implementation on a cloud testbed, which demonstrates the feasibility of engineering a high performance network search system.

II. Related Work

Most of the research in matching and ranking has been performed in the context of web search or search related to extensions of traditional web technologies. Many concepts that have been applied in web search and possibly refined later are based on earlier work in information retrieval (IR). IR does not generally distinguish between matching and ranking. Given a search term, an IR system produces a set of objects, each associated with a matching score. This score is often based on two metrics: the term frequency tf and the inverse document frequency idf [5]. Two examples of IR models that are relevant to this paper are the vector space model and the extended boolean retrieval model [3]. A brief discussion of IR concepts with respect to network search can be found in [1].

Traditional web search is performed on static web content and uses primarily link analysis for ranking, for example, PageRank [6] and HITS [7]. Current web search engines, which search static as well as dynamic content, use hundreds of metrics to compute ranking scores; in addition to page ranks, these metrics include a range of page usage statistics, matching scores from different IR models, freshness of data, etc. [8]. The weights of these metrics are generally determined using machine learning techniques applied to query logs [9], [10]. A widely used matching scheme in web search is BM25F, which is an extension of the popular IR model Okapi BM25 [11]. It is based on partitioning information on a web page into different fields, which are matched according to their importance.

With the evolution of Web 2.0, new functionality has been introduced, which gives rise to new metrics for ranking. For instance, in social tagging systems, metrics based on tags for pages and links influence the ranking score [12]; in live search systems, the freshness of the data influences the score [13]. Recently, web-based frameworks have been developed that provide access to specific types of information. For instance, the Web-of-Things framework, which supports Internet-of-Things technology, includes search functions that match against numerical values [14]. Second, the Web-of-Data framework makes available massive datasets in form of graphs. Term matching is performed against attributes associated with graph nodes [15], [16].

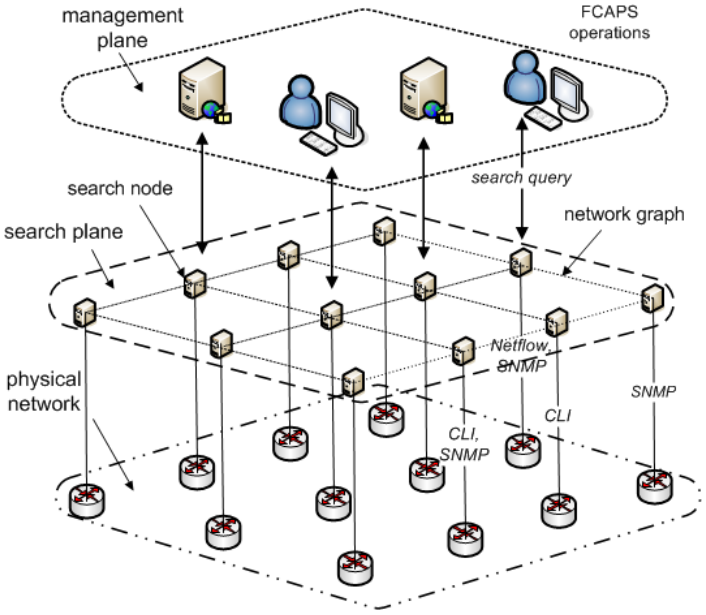

Fig. 1. An architecture for a network search system [1]

III. A Distributed Architecture for Network Search

Figure 1 shows an architecture for a network search system, which we introduced in [1]. Its key element is the *search plane*, which conceptualizes the network search functionality. This plane contains a network of *search nodes*, which have processing and storage capacities. A search node can communicate with a set of neighbors, which are identified through links of the network graph. The design of this plane supports searching in a distributed and parallel fashion. A search node can be realized in various ways: it can be part of the management infrastructure outside the managed system, it can be run as a standalone network appliance, or it can be integrated into a network element using a variety of technologies. Our current prototype implements the third option.

The bottom plane in Figure 1 represents the physical network that is subject to search. Each network element is associated with a search node, which maintains (or has access to) configuration and operational data from that network element. This data is modeled as a set of objects, whose structure is described in Section IV-A. Note that the figure shows the simplest form of association between a network element and a search node; it is possible that a search node maintains data from several physical devices, or, alternatively, a device updates data on several search nodes. The top of Figure 1 shows the management plane, which includes the systems and servers running processes for network supervision and management.

There are three important interfaces in this architecture. The first is the search interface, which supports the query language discussed in Section IV-B. We envision that every search node is an access point for search queries. The second interface defines the interaction

between a search node and a network element, which can be realized through polling or can be push based. This interface is technology-dependent and possibly proprietary. The third interface is the peer interface between the neighboring search nodes. It enables node to interact for distributed processing of search queries.

Each search node runs a process that communicates with the associated network element(s) from which it retrieves network data. A database function dynamically maps that data into the information model for network search and updates the local search database.

Search functions, invoked from the management plane through query invocation, are executed as distributed algorithms on the graph of search nodes. During the execution of a query on a search node, the local search database is accessed, the matching of the local query against stored indices is performed, and the local search result is possibly aggregated with results from other nodes.

IV. A SEMANTIC FOR MATCHING AND RANKING OF NETWORK SEARCH QUERIES

Following [17], our model for network search includes a four-tuple $\langle O, Q, M, R\rangle$. Here, O is the set of objects that represents the object space; Q is the set of syntactically correct queries on this space; M is the matching function, which determines, for a query, the set of objects in the query result; R is a ranking function, which orders the objects in the query result. In the following, we review the object model and the query language. Then, we introduce the matching and ranking functions for network search.

A. The object model

In [2], we presented an object model for network search. Objects in this model can represent physical and logical entities in a networked system, such as routers, servers, IP flows, virtual machines, etc. An object is modeled as a bag of attribute-value pairs—for short, attributes—whereby an attribute can contain configuration or operational information. An object has a unique name and a type, both of which are modeled as attributes. Figure 2 shows three sample objects. We say two objects are linked if they share an attribute.

B. The query language

In [2], we presented a query language for nework search, the core of which is given in BNF notation:

$$q \rightarrow \quad t \mid q \wedge q \mid q \vee q \tag{1}$$

$$t \rightarrow \quad a \mid v \mid a \ op \ v \tag{2}$$

$$op \rightarrow \quad = \mid < \mid \cdots \tag{3}$$

Rule (2) states that a token in this language is either an attribute name a, a value v, or an attribute-value pair a OP v. According to rule (1), a query can be a single token, or it can be constructed using tokens and boolean operators. We leave out here the discussion of the projection, aggregation and link operators of the language, because they are not essential to the discussion in this paper.

C. A semantic for matching search queries

In the simplest case, a matching function M maps a query and an object onto a boolean value, i.e., $M : Q \text{ x } O \rightarrow \{0, 1\}$. An object either matches a given query, in which case it is included in the result set of the query, or it does not match, in which case it is not part of the query result. We say that M is an exact matching function, if it maps to a boolean value. Such matching functions are generally used in databases. Also, the semantics of the query language we introduced in [2] is based on exact matching.

Web search does not use exact matching but approximate matching. In this case, the matching function M maps a query and an object onto a matching score in the interval 0–1, i.e., $M : Q \text{ x } O \rightarrow [0, 1]$. If M returns 0, the object is not included in the result set of the query; otherwise, it is included, and the value indicates the relevance of the object for the query: the higher the value the better the match.

Network search is similar to web search in the sense that search is often used to explore an information space, and the simple syntax of the query language does not always allow to express the intention of the invoker. Therefore, approximate matching should be supported in network search. Note also that exact matching is a special case of approximate matching.

In order to define the matching function M for approximate matching, we apply the extended boolean retrieval model, a popular IR model which was proposed by Gerard Salton et al. in 1983 [3]. The query language of this model is close to the one we are using for network search. An important part of the model is a similarity function, which measures the similarity between a query and an object, and which is analogous to the matching function M in the network search model.

Our function M uses two basic metrics from information retrieval, the term frequency tf and the inverse document frequency idf. In the network search model, these metrics relate to tokens of the query language. This means that the term frequency expresses the frequency of occurences of an attribute name, an attribute value and an attribute in an object, and the inverse document frequency indicates the inverse of the number of occurences of attribute names, attribute values and attributes in the object space. To give a specific example, consider the query with the token `IP-address`. The term frequency for this token and object (a) in Figure 2 is the number of occurences of this attribute normalized by the total number of attributes of the object, i.e., $tf = 0.333$. The inverse document frequency for this token relates to the probability that an object has the attribute *IP-address*. (We give here simplified version of the formulas for tf and idf in order to stress the basic ideas.)

For a specific object o, we compute M for a token t as $M(t) = tf_t \cdot idf_t$. Following [3], we define the function M for queries that are constructed out of n

object-name	:	urn:ns:cloud-08
object-type	:	server
cpu-core	:	12
memory	:	32 GB
IP-address	:	172.13.1.31
IP-address	:	172.13.1.32
IP-address	:	172.13.1.33
load	:	0.85
uptime	:	11350735.47
OS	:	Ubuntu-3.04

(a)

object-name	:	urn:ns:instance-07
object-type	:	VM
cpu-core	:	2
memory	:	4 GB
IP-address	:	192.168.1.5
server	:	urn:ns:cloud-08
load	:	0.5
customer	:	urn:ns:john:watson
uptime	:	350735.47

(b)

object-name	:	urn:ns:instance-18
object-type	:	VM
cpu-core	:	1
memory	:	2 GB
IP-address	:	192.168.1.19
server	:	urn:ns:cloud-07
load	:	0.4
customer	:	urn:ns:john:watson
uptime	:	650735.47
hypervisor	:	kvm

(c)

Fig. 2. Sample objects in a search space. (a) an object that represents a server in a cluster, (b)–(c): two objects that represent virtual machines in servers.

tokens and boolean operators as follows:

$$M(q_1 \vee \cdots \vee q_n) = \frac{||(M(q_1),\cdots,M(q_n))||_p}{\sqrt{n}} \quad (4)$$

$$M(q_1 \wedge \cdots \wedge q_n) = 1 - \frac{||(1-M(q_1),\cdots,1-M(q_n))||_p}{\sqrt{n}} \quad (5)$$

Equations 4 and 5 use the L_p vector norm, also known as P-norm. Choosing $p = \infty$ results in M being the matching function of the (conventional) boolean model in IR, i.e., M performs exact matching [18]; choosing p in the interval $[1, \infty)$ results in an approximate matching function. Changing p to a smaller number increases the number of matched objects for a given query. For $p = 1$, M becomes the matching function of the vector space model in IR and performs loose approximate matching [4]. With equations 4 and 5, the matching function on any boolean expression is well-defined.

In our matching model, all tokens have the same weight. The framework in [3] upon which we base our matching function allows us to include weights for tokens in a straightforward way. We plan to study such an extension as part of our future work. (Models for term weights are important for web search systems.)

D. A ranking function

A ranking function R maps the result set of a query q onto an ordered list. The result set O_q of query q includes those objects in the object space whose matching score is positive, i.e., $Q_q = \{o \in O | M(q, o) > 0\}$. The first element of the list is the object that is most relevant to the query, and the relevance decreases with each subsequent list element. In search systems, this list is often truncated after k elements in order to limit the size of the result set. Note that the matching function M introduced in the above Section IV-C is a ranking function. In the following, we will extend M to include heuristics that are relevant to network search, and we will express the relevance of an object to a query in form of a ranking score.

The first metric we are considering measures the similarity between an object and the query, the second relates to the connectivity of an object within the graph of all objects, whereby links between objects express relationships through joint attributes (see Section IV-A), and the third relies on information freshness. In the search literature, such metrics are also called weights or signals.

First, our similarity metric uses the matching function M discussed above, extended to accomodate our object model. The matching rule for an object name is extended, so that a token matches a substring in a name, e.g., 'john' matches 'urn:ns:john:watson'. Also, the contribution to the matching score is higher for the name and the type attributes than for other attributes in an object, since we consider objects matching a query via name or type more relevant to the query than objects matching the query via other attributes. Second, the ranking score considers the connectivity of an object within the graph of all objects. The intuition behind this metric is that a high connectivity of an object signifies a high importance of this object. In web search, the same idea of measuring connectivity is behind the page rank or hub score metrics [19]. Third, the ranking score considers the freshness of the information contained in an object: the more recent the information, the higher the score, which means that fresh information is more relevant to a query. Finally, note that other metrics can be considered when computing the ranking score, for instance, metrics relating to location or search history of a query invoker. Such metrics have proven useful in other search systems, and we plan to study them for possible inclusion in our network search system. We compute the ranking function R as a weighted sum of the above metrics.

The matching and ranking functions discussed in this paper compute, for a given query q and object space O, the result set and an ordering of objects within the set. The position of an object within this ordering expresses its relevance with respect to the query. It is important to recognize that this relevance is highly dependent upon the management task that uses network search to obtain information from the network. For instance, consider three different management tasks: a human operator who "googles the network" to identify the root cause of a fault, a cloud management application that performs virtual asset tracking, and an anomaly detection application that searches for abnormal patterns

in network state information. For the same or similar queries, all these applications may consider different result sets as appropriate and different ordering policies to reflect their respective relevance. For this reason, matching and ranking modules in a network search system must be generic, so that they can be instantiated for specific application purposes. In our current design and system implementation, the matching function can be initialized with different term weights and different values for the p vector norm ($p = \infty$ for exact matching, $p < \infty$ for approximate matching, and the default is $p = 2$). The ranking function can be initialized with different weights for the metrics discussed above. A weight can be set to 0 to ignore a specific metric.

V. Distributed processing of search queries

Our approach to process network search queries makes use of the echo protocol, a tree-based protocol suitable for distributed polling [20] [21]. It is based on an algorithm first described by Segall [22]. The echo protocol executes on the network graph of the search plane (Figure 1). It can be started on any search node once a query q has been received. First, the query is disseminated to every node and executed against the local database D. The results of all local operations are sent along a spanning tree, where the partial results are aggregated. The definition of the local operation, the aggregation operation of the query result, and the current local state of the query collection, are modeled in an object, called the *aggregator object* of the echo protocol.

Figure 3 shows the aggregator object for distributed processing of queries expressed in the language given in Section IV-B. The object contains the functionality for local data retrieval, matching and ranking, as well as for incremental aggregation of the partial results along a spanning tree. The state of the aggregator object is captured in the variable qr, which stores the local search result of query q, potentially also an aggregated search result. qr is of type dictionary, which contains 3-tuples of the structure (object name, object, ranking score). Each tuple contains an object retrieved from the local database, together with its ranking score relative to query q. The dictionary function *insert()* inserts a tuple into a dictionary, the function *merge()* combines two dictionary into a single one, and the function *top-k()* creates a dictionary with the top k tuples, according to decreasing ranking score, of a given dictionary. The procedure *local()* in Figure 3 initializes the aggregator state, executes the query q against the local database D, implicitly through invoking the matching function M, stores the query results in qr, which includes invoking the ranking function R, and reduces qr, if needed, to include the top k objects only. The procedure *aggregate()* aggregates the local state qr with the state $child - qr$ from a child node by merging the two dictionaries and reducing the size of the resulting dictionary if applicable. This procedure implements the process of the distributed aggregation by the echo protocol. Figure 3 contains the partial pseudocode of the aggregator object. For instance, the processing of

```
1: aggregator object processQuery( )
2:     var: qr : dictionary;
3:     procedure local( )
4:         qr := { };
5:         for each o ∈ M(q,D) do
6:             insert (name(o), o, R(q,o)) into qr;
7:         qr := top-k(qr);
8:     procedure aggregate(child-qr: dictionary)
9:         qr := top-k(merge(qr, child-qr));
```

Fig. 3. Aggregator for processing a query q on a node with local database D.

operators of the full query language [2], including, the projection, aggregation, and link operators, is missing. An aggregator object that processes these operators is given in [2]. Second, the code in Figure 3 does not reflect the fact that the matching function M and the ranking function R are parameterized (see Section IV-D). These parameters must be passed together with the query q to the aggregator object. Third, both M and R rely on object metrics, which must be retrieved from the local database. Two of these metrics, idf for each attribute and $connectivity$ for each object, are global and are computed using a global aggregation protocol, which runs independent from query processing.

The performance properties of the echo protocol [20], which determines key performance metrics of the query processing scheme, suggests that the presented generic approach to network search is a scalable solution. For instance, the execution time of a query grows proportionally with the height of the spanning tree, which is upper bounded by the diameter of the network graph. The protocol overhead is evenly distributed on the network graph, as two messages traverse each link during the execution of echo. Lastly, the number of messages each search node processes is upper bounded by the degree of the network graph.

VI. Design and implementation of a search node

The search node is the key architectural component of a network search system (Figure 4). All search nodes are identical in functionality and co-operatively provide the network search service. A search node has an interface to the management plane, which provides an access point for network search, it executes distributed query processing (echo protocol, local data access, matching, ranking, and aggregation), it maintains a real-time database with network objects, and it includes a sensor subsystem that populates this database. The performance goals for designing a search node are (a) low latency for search queries, for supporting real-time search requirements; (b) low computational overhead for query processing, since search nodes may be hosted by service devices; (c) high throughput of search queries, since we expect a high number of concurrent queries in a large system; (d) support for a large number of network objects (larger than 10,000), to make available to network search an extensive set of operational data.

Figure 4 shows the main components of a search node and their interactions. A search node has three interfaces: to the management plane, to peer nodes of the search plane, and to associated devices (possibly internal to a device) of a networked system. The component on top of the figure is dedicated to query processing. It interacts with the management plane, peer nodes and the local database. The component on the bottom includes the sensing functionality that updates the database.

The component in the middle of the figure contains the database module. Our design calls for a document oriented NoSQL database with object-level access [23]. Such a database allows us to implement our object model in straightforward way, and it supports the processing of search queries by providing the basic functionalities for token matching through the database query interface. The object database is complemented with an index structure with entries of the form (key, object id, matching metric, ranking metric). The key field refers to a token in the query language, object id is a pointer to the object in the database, and the matching and ranking metrics contain information needed by the matching and ranking functions for query processing. The index significantly increases the performance of token matching, at the expense of maintaining it.

Distributed query processing is achieved by local processing and exchanging of messages between search nodes. Figure 5 shows our design for distributed query processing. Each circle (in the bottom of Figure 5) represents a thread that executes asynchronously. The distributed design allows us, on a multicore hardware, to achieve a higher throughput for processing search queries and a lower query latency compared to a straightforward single-thread design. On top of the figure, we find buffers for incoming and outgoing messages, two buffers for each peer node or process in the management plane. A message router retrieves the messages from the message in-buffers and places them in the input queues of the query processors. Messages that relate to the invocation of the same query, i.e., which contain the same invocation identifier, are assigned to the same processor. A query processor retrieves a message from its input queue and executes a query processing step. Such a step includes updating the state of the query invocation, for instance, updating the states of the echo protocol and its aggregator, and executing the procedures in the aggregator, for instance the procedure *local()*, which accesses the local database. As part of the query processing step, one or more messages are generated, which the query processor inserts into the output queue. Another message router places messages from this queue into the appropriate message out-buffers. The optimal number of query processors in a system configuration is dependent on the hardware platform of the search node. In addition, it depends on the database capacity and the amount of CPU resources that can be devoted to network search.

We implemented the design of the search node on a multicore architecture. All components, except the database system, are written in Python. We use

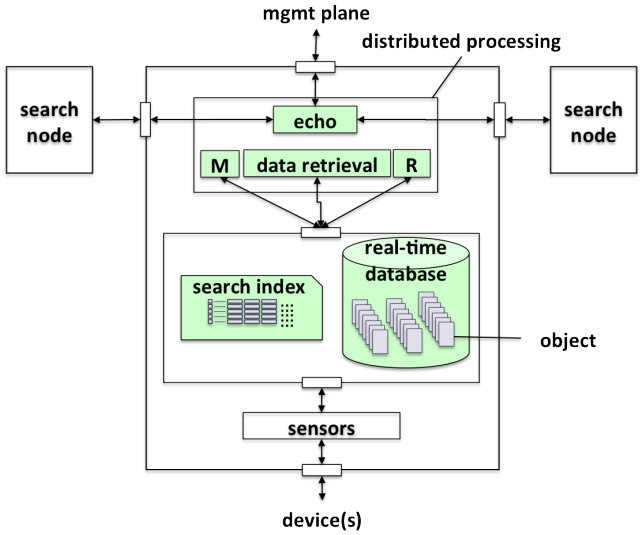

Fig. 4. Architecture of a search node

the multiprocessing package for threading on a multicore hardware. The database component is based on MongoDB [24], a popular opensource database system. MongoDB is a document-oriented NoSQL system that realizes a persistent database. (It includes supports for distributed databases, which we are not using.) We chose a document-oriented database system over a key-value store package, because of the object-level abstractions that a document-oriented database supports. MongoDB allows us to implement our object model in a straightforward manner, supports basic token matching, and exhibits good performance for read and write operations, compared to other document-oriented databases. While MongoDB is not an ideal database system for our purposes—we would prefer an in-memory database with attribute-level locking—, it seems to us currently the best choice available. Our current prototype has 5000 objects per search node. The query processing component runs five threads on four cores (message routers share a core), and the database component runs on a single core.

The above design of a search node reflects our performance goals. The purpose of maintaining an index structure is to increase query throughput and lower query latency. The distributed design of local query processing has the same objectives. It also allows us to control the computational overhead of query processing by choosing the appropriate number of cores for this task.

VII. Evaluation of a Network Search Prototype on a Cloud Testbed

We have instrumented the servers of a cloud platform for network search. The platform includes nine high-performance servers, interconnected by Gigabit Ethernet, and runs the OpenStack cloud management software. (See [25] for details.) Each server includes a search node. The real-time database on a search node has currently four types of objects, namely, server,

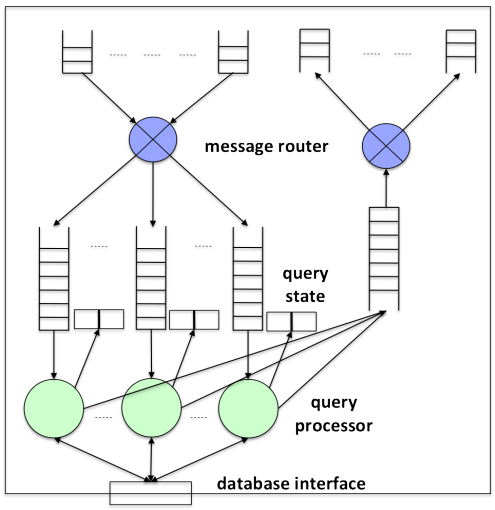

Fig. 5. Concurrent query processing on multicore hardware

virtual machine (VM), application, and customer. There are approximately 5000 objects per search node, where most objects have 20 to 30 attributes. A data sensing component reads system files, such as '/proc' [26], libvirt configuration [27] etc., and populates and periodically updates the objects in the local database at a rate corresponding to their respective lifetime. The network graph that defines the peer relationships between search nodes has small world properties and is currently statically configured. (Each search node has between two and four neighbors.) For all experiments, the matching function in query processing is configured with $p = 2$, and the ranking function returns the top 100 objects. Management stations that create query load are run on lower-performing servers of the testbed. A demonstration of this system is presented in [28].

We produce synthetic load for the performance measurements as follows. The query load consists of global search queries with 2 to 5 tokens each. Tokens are either attribute names, values or attribute-value pairs, with equal probability. The values for the tokens are chosen uniformly at random from the global object space. The number of tokens per query is 2 to 5 with equal probability. Queries are injected using a Poisson process and are sent with equal probability to each search node on the testbed. Note that the global query load is the same as the local query load on a search node, since each query is executed on all servers (see Section V). During an experiment, a search node processes a mix of 75 percent global queries and 25 percent local updates. The Local update load consists of insert object, delete object and refresh attribute operations, with equal probability. The objects and attributes are selected from the local database with equal probability. New objects are randomly created from a schema of four object types. Update operations are invoked on the local database following a Poisson process.

During an experiment, we measure two metrics: first, the latency of each global query, measured from the time the management station sends out a request until it receives the response; second, the CPU utilization on each server that runs a search node.

Before measurements are performed, the real-time databases on the search nodes are initialized using a script and synthetic data. For each run of an experiment, we inject query/update load at specific rates, wait until the system is in steady state, generally around 10 seconds, and take measurements, over a period of some 120 seconds. The query load for the experiments ranges from 25 queries/sec to 700 queries/sec.

We report on three sets of experiments. First, we investigate the latency of global queries in function of the query load. Figure 6 shows the measurement results of runs with query loads ranging from 50 milliseconds to 450 milliseconds. As we expect, both the median latency (more generally, the 25th, 50th, 75th, and 95th percentile), as well as the deviation of the latencies increase with increasing load. Further, up to a load of 200 queries/sec the deviations of the latencies are quite small, with latencies for the 75th percentile well below 100 milliseconds.

Second, we investigate how the number of query processors on dedicated cores affects the query latency. Figure 7 shows four curves, each one relating to a series of experiments for a number of query processors ranging from one to four. The curves show the median latencies. The curve with the three concurrent processors is based on the same measurements as Figure 6. We observe that, for all curves, the query latency increases with load. Beyond a certain load, the latencies rise steeply, which we explain with approaching the capacity of the query processing system. We also observe, for a given latency target, say 200 milliseconds, that the system capacity increases with the number of query processors. Beyond a certain number of processors, three processors in our configuration, the gain in system capacity decreases, an effect we explain by scheduling queries from an increasing number of processors towards a single database access point.

Third, we investigate the computational overhead of network search. Figure 8 shows the CPU utilization of the servers produced by the search nodes in function of the query load, for different number of query processors. As above, the figure shows four curves, for different numbers of query processors. The curves are based on the same measurement results as those used by Figures 6 and 7. Each curve shares a linear segment and flattens out at some query load. We attribute the sudden change of slope in a curve to the query processors becoming overloaded (which we confirmed through other measurements). Recall that we run the experiments on 24 core processors, where each processor can consume up to 4.1% of CPU capacity. The experiment shows that controlling the number of query processors is an effective way of controlling the overhead of the network search system.

The above measurements demonstrate that our prototype can support a load of 200 queries/sec at a latency below 100 milliseconds for the 75th percentile of the

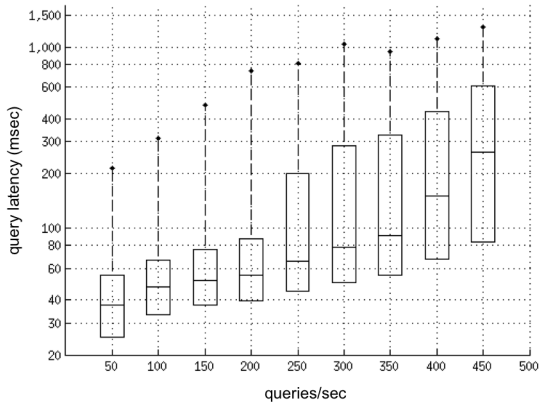

Fig. 6. Latency of global queries for different query loads. Each measurement shows the 25th, 50th, 75th, and 95th percentile value. The node runs three concurrent query processors.

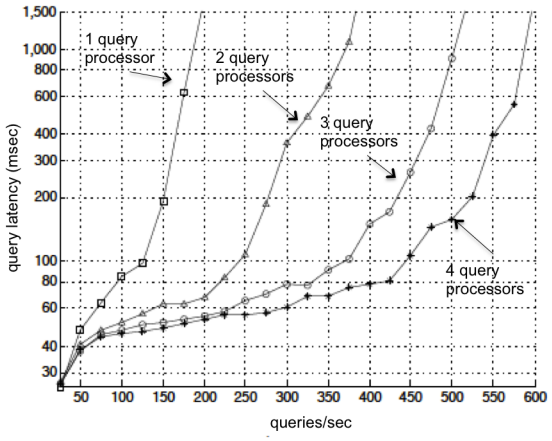

Fig. 7. Impact of concurrency of local query processing on median latency of global queries for different query loads. The curves represent configurations for 1,2,3,4 query processors, each processor execution on a dedicated core.

queries, by using at most 5% of the CPU, when running three query processors on three dedicated cores. Based on the properties of the echo protocol, we conclude that when the system is scaled up to, say, 100,000 nodes, the above performance metrics will stay approximately the same, except for the latency, which will increase.

VIII. Discussion

In this paper, we have proposed a set of parameterizable matching and ranking functions for a simple query language that we developed for network search. We have shown how matching and ranking functions can be computed in a distributed and scalable way. Further, we presented the design of a network search node and reported on measurement results from a network search prototype on a cloud testbed.

The semantics for matching and ranking introduced in Section IV-C and IV-D are tailored to networked

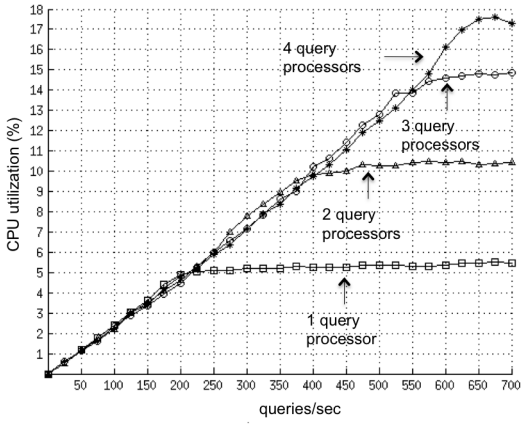

Fig. 8. Computational overhead of network search: CPU usage of search node for different query loads and number of concurrent query processors.

systems, but not to specific technologies. For instance, we propose specific matching rules for name resolution or ranking policies that consider freshness of objects. However, up to now, we did not consider specific matching rules for IP networks, for example, although such rules can support powerful explorative search. For instance, given an IP address as a search term, NAT-aware address matching could match the address to its translated version in another domain. Furthermore, a matching function that exploits the concept of an IP address could match an address to a subnet or vice versa. Also, a flow id could match another flow identifier that belongs to the same application. Lastly, the name of a computing device, for instance, could match data representing the DHCP server that provides the address for the device, or it could match the AAA rules that define the security policy of the device.

The measurements from our prototype system show that it is feasible to build a network search system that can process a load of 200 global queries/sec with an overhead of less than 5% CPU load on our cloud platform. Knowing the design of the system and the properties of the echo protocol that underlies query processing, a back-of-the-envelope calculation shows that a system of 100,000 servers processing the same load would exhibit the same overhead per server and a query latency of below 1 sec.

Up to now we have developed a functionally complete, simple design of a network search system. Much work remains to be done for our design to be effective in practical scenarios. For instance, security and privacy issues need to be addressed, concepts for search space reduction need to be developed, search across multiple domains needs to be investigated, etc.

References

[1] M. Uddin, R. Stadler, and A. Clemm, "Management by network search," in *IFIP/IEEE Network Operations and Management Symposium (NOMS 2012), Maui, Hawaii*, April 16 - 20, 2012.

[2] M. Uddin, R. Stadler, and A. Clemm, "A query language for network search," in *IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium*, May 27-31, 2013.

[3] G. Salton, E. A. Fox, and H. Wu, "Extended boolean information retrieval," *Commun. ACM*, vol. 26, pp. 1022–1036, Nov. 1983.

[4] G. Salton, A. Wong, and C. S. Yang, "A vector space model for automatic indexing," *Commun. ACM*, vol. 18, pp. 613–620, Nov. 1975.

[5] C. D. Manning, P. Raghavan, and H. Schtze, *Term frequency and weighting*, ch. Scoring, term weighting and the vector space model, Introduction to Information Retrieval, pp. 117–119. Cambridge University Press, 2008.

[6] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, pp. 107–117, Apr. 1998.

[7] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, pp. 604–632, Sept. 1999.

[8] M. Andrews, "Searching the internet," *IEEE Software*, vol. 29, pp. 13–16, 2012.

[9] N. Fuhr and C. Buckley, "A probabilistic learning approach for document indexing," *ACM Trans. Inf. Syst.*, vol. 9, pp. 223–248, July 1991.

[10] T. Joachims, "Optimizing search engines using clickthrough data," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '02, (New York, NY, USA), pp. 133–142, ACM, 2002.

[11] S. E. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[12] A. Gulli, S. Cataudella, and L. Foschini, "Tc-socialrank: Ranking the social web," in *Algorithms and Models for the Web-Graph* (K. Avrachenkov, D. Donato, and N. Litvak, eds.), vol. 5427 of *Lecture Notes in Computer Science*, pp. 143–154, Springer Berlin Heidelberg, 2009.

[13] C. Chen, F. Li, B. C. Ooi, and S. Wu, "Ti: an efficient indexing mechanism for real-time search on tweets," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, (New York, NY, USA), pp. 649–660, ACM, 2011.

[14] B. Ostermaier, K. Romer, F. Mattern, M. Fahrmair, and W. Kellerer, "A real-time search engine for the web of things," in *Internet of Things (IOT), 2010*, pp. 1 –8, 29 2010-dec. 1 2010.

[15] A. Tonon, G. Demartini, and P. Cudré-Mauroux, "Combining inverted indices and structured search for ad-hoc object retrieval," in *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '12, (New York, NY, USA), pp. 125–134, ACM, 2012.

[16] R. Neumayer, K. Balog, and K. Nørvåg, "On the modeling of entities for ad-hoc entity search in the web of data," in *ECIR*, pp. 133–145, 2012.

[17] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, May 1999.

[18] A. Singhal, "Modern information retrieval: A brief overview," *IEEE Data Eng. Bull.*, vol. 24, no. 4, pp. 35–43, 2001.

[19] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets*, ch. 14, pp. 397–435. Cambridge University Press, 2010.

[20] R. Stadler, "Protocols for distributed management," Tech. Rep. 2012:028, KTH, Communication Networks, 2012. QC 20120604.

[21] K. Lim and R. Stadler, "A navigation pattern for scalable internet management," in *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pp. 405–420, Ieee, 2001.

[22] A. Segall, "Distributed network protocols," *IEEE Transactions on Information Theory*, vol. 29, pp. 23–35, 1983.

[23] P. Sadalage and M. Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, ch. 2, p. 20. Addison-Wesley, 2012.

[24] "MongoDB Manual." "http://docs.mongodb.org/manual/", August 2012.

[25] F. Wuhib, R. Stadler, and H. Lindgren, "Dynamic resource allocation with management objectives : Implementation for an openstack cloud," Tech. Rep. 2012:021, KTH, Communication Networks, 2012. QC 20120528.

[26] "proc." "http://manpages.courier-mta.org/htmlman5/proc.5.html", August 2012.

[27] "libvirt 0.7.5 - Application Development Guide." "http://libvirt.org/guide/html/", August 2012.

[28] A. Uddin, M. Skinner, R. Stadler, and A. Clemm, "Real-time search in clouds," in *IFIP/IEEE Integrated Network Management Symposium (IM 2013), Demonstration Program, Ghent, Belgium,*, May 27 - 31, 2013.