Sponge: A Searchable P2P Mobile App Store using DHTs

Md. Khaledur Rahman
khaled.cse.07@gmail.com

Md Yusuf Sarwar Uddin
yusufsarwar@cse.buet.ac.bd

Nashid Shahriar
nshahriar@cse.buet.ac.bd

Mostafizur Rahman
topucse06@gmail.com

***Abstract*—In this paper we propose a novel searchable P2P content storage for storing mobile apps. In the growing need of storing and distributing huge number mobile apps across large number of users, it has been deemed necessary to think of a storage that leverages enormous amount of content space that common people can share these days. In our proposal, we leverage popular P2P stores, such as standard DHTs, for storing apps. One problem with DTHs is that they support flat names (denoted as keys) for content objects, which do not make content objects *searchable* by their different attributes. When users look for a certain app in an app store, they do not necessarily look by their unique names, instead by a set of attribute-value pairs (multi-attribute queries). We devise a technique, called *predicate hashing*, by which we hash these attribute-value pairs into hashed keys that are in turn used to locate corresponding apps from the underlying DHT. By doing this we effectively instrument flat named DHTs into reverse ifndexable content store. To this end, we build Sponge, a *Kademlia* based P2P app store, and do simulation based experiments to show that Sponge can retrieve content against multi-attribute queries.**

I. Introduction

In this paper we propose a searchable P2P content store for storing software applications (a.k.a, apps), particularly mobile apps. Over the past half a decade, mobile apps have become significantly popular because of a reasonable revolution in smart phones. And the demand is increasing day by day. Usually applications are served to the users from a common centralized storage, which are known as an App Store. An app store is generally a form of an online store, where users browse through different categories and genres of applications, view information and reviews related to apps, purchase the apps (if necessary and required so), and then automatically download and install the application on their devices. There are many online stores which serve mono platform based app such as Google play [1] for Android platform, iTunes [3] or App Store [4] for iPhone platform and so on. There are also some cross-platform based stores which serve application software for various platforms such as Handmark [5], Opera store [6], etc. But these cross-platform based stores do not have a large number of application software. While cross-platform stores may not have a large number of apps, popular app store has gigantic number of apps (Google Play has nearly 1.3 million apps [2]) requiring a huge amount storage.

In the growing need of storing and distributing huge number apps, particularly mobile apps, across a large number of users, it has been deemed necessary to think of a storage that can leverage enormous amount of content space that common people can share these days. Peer to peer (P2P) systems allow such leverages. In that, participant nodes dedicate some of disk space to form a P2P app store. A P2P app store can be motivated in several folds. Firstly, due to distributed in nature a P2P app store would be highly scalable and available that are fundamental requirements for app store, where a large number of developers put their content into it and a larger number of users download those apps. Secondly, developers who produce and push apps in the store can easily dedicate their disk space for this system to be built and they have direct incentives for doing so. Unlike single platform popular app stores (such as Google Play for Android), a P2P app store can host apps for all possible platforms (such as Android, iOS, Windows and Blackberry). For higher dissemination, developers indeed create the same app for different platforms so the developer can publish them to the same store and maintain them easily. Finally, a large number of apps are usually free so shared nature of a P2P store matches the economic mode[1].

Large scale P2P storages are usually implemented as DHTs (distributed hash tables). So DHTs could be a natural choice for the app store. One problem with using DHTs is that they support only at names (denoted as keys) for content objects, which do not make content objects searchable by their different attributes. When users look for a certain app in an app store, they do not necessarily look by their unique names, instead by a set of attribute-value pairs usually given as a high level description, e.g., "talking tom for Android tablet". Unlike other P2P based file and content sharing system, such as web cache [8], P2P citeseer [9], where contents are retrieved by application generated unique names, an app store must have *in situ* search capabilities to find applications satisfying certain criteria, specified by a search query. These search queries, although are usually obtained as high level strings from users, are translated into a set of attribute-value pairs for an appropriately defined set of attributes. For example, a user query for "talking tom" originated on an Android tablet can be translated into {name="talking tom", category="game", platform="android", device="tablet" }. We hence assume that search queries are expressed mainly as a set of attribute-value pairs, called as predicates.

[1]Even though apps are not free, arrangement can be made for the corresponding developer to earn the fee. Sharing the platform cost could be tricky though.

In our proposal, we leverage popular P2P stores, such as standard DHTs, for storing apps. The core technique we use is called predicate hashing in which everything, that is data objects (i.e., apps) as well as query strings, is converted into hash keys that are later inserted and looked up in the DHT overlay formed by the participant nodes. The system only relies on the very fundamental service available in all DHTs, namely **`lookup(key)`**, that locates the node labeled by an identifier, 'key' (or the node responsible for representing this 'key'). Once the node is obtained, appropriate RPC methods can be invoked in order to perform specific operations on that node, namely **`insert`** and **`search`** operations. While hashing data objects are rather easy (done by their unique UUIDs or keys), query predicates need special attention: attribute-value pairs in the predicate are hashed to generate keys that are in turn used to locate corresponding apps from the underlying DHT. By doing this, we effectively instrument the at named DHTs to become a reverse indexable content store. One of the strengths of the scheme is that we do not assume any specific DHT in place, rather any DHT, such as Chord [13], Pastry [14] and Kademlia [18], works alike. In our implementation, we however use Kademlia.

To this end, we propose Sponge, a distributed P2P app store. In this system, peer nodes arrange themselves in a DHT. Apps are stored at a location hashed by their keys (key is an opaque bit string that uniquely identifies each individual app). Meta information about an app, which is constructed as a list of attribute-value pairs, is also stored at certain locations (again hashed by the attribute-value pairs in a certain fashion). Query strings are also routed to specific nodes (obtained through hashing over the string) so that objects satisfying the query strings can be retrieved from those nodes. In our scheme, we consider apps as bundle of binary data payloads described by a set of attribute-values that are signed and ready to be installed on devices once downloaded from the content store. In that, we use the terms apps, objects, and content interchangeably.

The rest of the paper is organized as follows. In Section II, we will describe the model and architecture of our system. We will describe query engine in Section III followed by the experimental setup and results in Section IV. Then we conclude briefly in Section V.

II. Model and Architecture

Our system provides a multi-attribute based query solution for hierarchically clustered environments. In this section, we describe the basic working of Sponge followed by a description of attributes and meta data model.

A. Basic Working of Proposed System

Sponge constitutes a DHT of participating nodes, called peer nodes. There are two kind of users in Sponge, namely those who "put" content into the app sotre (called *clients*), and those who "get" apps (called *users*). When a developer (client) wants to deploy an app to the store, he needs to provide some information about the app. We use these information as attributes of the data object (app). The provided list of attributes are then used to find a list of suitable locations (nodes) in the DHT in order to store that object (app content) onto them. An user may search for an app to download and install on his/her device. A high level description of the query provided by user is transformed to a list of attribute-value pairs. This transformation is, however, beyond our scope; we only assume such transformation exists. A distributed search is then initiated across the overlay. The primary objective of the query is to locate those peers that hold the requested objects. Sometimes this query is mapped to exactly one app or sometimes to a list of apps. So, our system responds providing a list of apps (by their names or unique Ids), but not the original content themselves. When a user chooses an app to download and install, only then the original app content will be served to the user (i.e., retrieved from the DHT). A measurable amount of messages will be passed in the process while inserting objects to the store and also while searching and downloading apps from the store.

B. Types of Attributes

When a developer wants to deploy an app to the store, s/he will need to provide some information about the app. Sponge transforms the provided information as attributes of the app. An app can be uniquely identified by using a certain combination of these attributes. In our system, there are four types of attributes namely *Searchable*, *Hashable*, *Displayable* and *Mutable*. *Searchable* attributes are those which are used to search an app or a collection of apps from the store. These attributes are defined during uploading the app content only once and are not changed later such as *name, platform, category* etc. *Searchable* attributes are visible to users. *Hashable* attributes are those attributes which are used to locate nodes in the DHT (by hashing on them). All hashable attributes are searchable, but the converse is not true. For example, for a query "games released after 2012", attribute, **`category`**, is hashable, but **`releaseyear`** is not. Later we shall show attributes to take a certain exact value (rather than a range) is hashable. *Comparable* or *Displayable* attributes are those attributes based of whose values result set of a query is ordered, for example **`rating, popularity, version,`** etc. *Mutable* attributes are those which can be changed with time, such as *popularity, rating, downloadcount,* etc. These attributes are mostly updated through on the user's feedback. We can argue that searchable attributes are not mutable.

C. (Meta) Data Model

An object is defined $\{id, (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n), payload\}$, where v_i is the value of attribute a_i and key is the preprocessed value of attribute list, and $payload$ is the actually binary content of the app. Except the $payload$, rests are assumed as meta data of the object.

The meta data modeling is an important part of the system. Efficiently serving to a query depends much on it. Such as $size$ cannot be stored as meta information. Because, users don't try to make a search giving the $size$ of the app. App description, normally the attribute list provided by developers and some preprocessing values of it are treated as meta information about the object. Our system stores meta information for each app as a set of (attribute, value) pairs. A popular app named 'DX Ball' can be expressed as: $\{(category$, Games), ($subcategory$, Arcade), ($releaseyear$, 2014), ($platform$, Android), ($version$, 1.1.1), ($language$, English), ($country$, Bangladesh), ($device$,

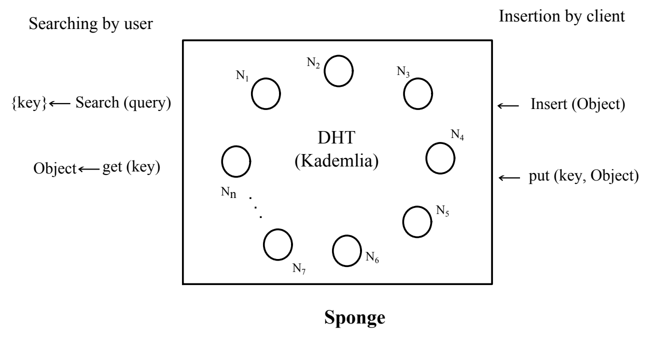

Fig. 1: Sponge architecture. N_i denotes a node for different value of i.

Mobile)} along with {($downloadcount$, 0), ($rating$, 4.0), ($size$, 2MB)}. Here, $downloadcount$ and $rating$ are ***Mutable*** attributes and can be used to order the popular apps or display the search results in some particular order as discussed in II-B. In any presentation of the list of attributes (whether in *objects* and in *query strings*) we always assume that attributes are ordered alphabetically.

D. Overlay Construction

At the very beginning of execution, we have fixed the total number of nodes participated in the system. No new node will be connected to the system. But some nodes may go down and later they can also join the network. In the network topology, we consider that every node is connected to the Internet and forms a standard $kademlia$ network as in Figure 1. They can communicate with each other through the Internet. It will be easy to maintain query engine of our system in such network topology. An object can be stored via any node of the system. During the insertion, system automatically makes some copies of it's meta information to store it in specific number of peer nodes. But the original content is stored in one location which is retrieved only when user tends to install the app. Similarly a search query can be made at any node. Sponge automatically initiate a distributed search and does the rest of the post processing works to retrieve results.

III. Query Engine

Sponge has been designed to provide multi-attribute based query solution. Once the system is initiated, it becomes ready to store objects and serve queries. Original object is stored in the content node and meta information about that object is fused in the peer nodes in the form of query overlay. In the next two subsections, we describe how data items are stored in Sponge and queries are served to the users.

A. Insertion of Object

When someone wants to insert (upload) an object (data item, d) to the sytem, s/he will provide a list of attributes $(a_1, a_2, a_3, \ldots, a_n)$. Algorithm 1 describes the insertion operation of an object to the system. Here, clients invoke **`insertObject(object)`** procedure. On the other hand, Sponge node invokes **`insert(key, object)`** and **`insertMetadata(key, object.metadata)`** procedures. These procedures are called through RPC.

Algorithm 1 Insertion of an Object

```
$object = \{key, (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n), payload\}$

insert(key, object){
$contentstore[key] = object$
}

insertMetadata(key, object.metadata){
// mstore is a metadate storage
mstore[key].Add(object.metadata)
}

insertObject(object){
$id = DHT.hash(key)$
$node = DHT.lookup(id)$
$node.insert(key, object)$
for binary representation of $i$ from 0 to $2^n - 1$ do
  $list = \emptyset$
  if number of '1' in $i <= k$ then
    $mask = 2^{n-1}$
    for $k$ in 1 to $n$ do
      if $i$ & $mask != 0$ then
        $list = list \cup (a_i, v_i)$
      end if
      $mask = mask >> 1$
    end for
    $id = DHT.hash(list)$
    $node = DHT.lookup(id)$
    $node.insertMetadata(key, object.metadata)$
  end if
end for
}
```

First of all, the system performs a consistent hash function on key provided in object description and generates a node id to store the object. Sponge initiates a standard Kademlia DHT **`lookup(id)`** to locate the peer node responsible for this 'id'. This is location where the app is stored. Sponge then tries to fuse meta information of this object across a set of peer nodes based in its attribute-value pairs. In this, Sponge actually attempts to pseudo-produce a set of query predicates (each being a list of attribute-value pairs) that this particular object can satisfy. These queries are nothing but several combination of various attribute value pairs of this very object, because a search with these attribute-value pairs should return this object. Since generating all combinations can be expensive (exponential in size), the system uses combinations upto a certain size, k. In that, predicates with 1 attribute (out of total of n), combinations with any 2 attributes, combinations with any 3 attributes, thus upto k attributes are generated. So a total of $\binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{k} = \sum_{i=1}^{k} \binom{n}{i}$ predicates are generated.

Each of the generated predicate is then hashed (described) to find the corresponding DHT nodes where the object's meta information is stored. Note that at this stage, only the meta information, not the original binary payload, is stored at those nodes. When any of these query predicates is made by users later on, one of these nodes would be in turn reached, thus retrieving the corresponding objects satisfying the query. Note that metadata of a single object is stored at several different nodes, which gives high robustness in retrieving objects amid of network dynamics, such as node failures.

Algorithm 2 Searching Object

```
Require: search string: S
         search query: $s_1, s_2, s_3, ..., s_n \in S$

  $query = \{\mathcal{P}, limit\}$
  $\mathcal{P} = \{l_{av}, l_{ar}, l_{ad}\}$
  $l_{av} = (attribute, value)$ pairs (hashable attributes)
  $l_{ar} = (attribute, range)$ pairs (searchable attributes)
  $l_{ad} =$ list of displayable attributes
  $limit =$number of items to be returned

  search(query){
  // items having exact match with attributes
  $items = mstore.LookupItems(query.list_{av})$
  $result = \emptyset$
  for item in items do
    if $(satisfy(item, list_{ar}))$ then
      result.add(item)
    end if
  end for
  //sort items in results based on attributes in $list_{ad}$
  return keys of result[0:limit]
  }

  retrieveObject(key){
  return contentstore[key]
  }

  searchObjects(query)
  $query = (a_1, v_1), (a_2, v_2), \ldots, (a_m, v_m)$
  if $m <= k$ then
    $node = DHT.lookup(DHT.hash(query))$
    $result = node.search(query)$
  else
    // Take first $k$ attribute value pairs from query
    // let this be $q_k$
    $node = DHT.lookup(DHT.hash(q_k))$
    $result = node.search(query)$
  end if
  return result
```

B. Serving Queries

Once the data items are inserted, our system is ready to serve queries. The system is designed to serve for both single attribute and multi attribute queries. When an user searches for app(s) in the store, he provides a query predicate a distributed search is initiated in the system. Algorithm 2 describes the searching of object(s) in the system.

Every query is expressed as a predicate, $\mathcal{P}$, over an attribute set of the requested objects. In that, each query would specify specific requirements over a set of attributes that the requested content objects should satisfy. These specification can be in form of an attribute-value pair which asks a certain attribute to be of a certain value (i.e., using "equal to" operator), or it can be using any other operators, say relational operators. In general a query predicate, $\mathcal{P}$, can be expressed as a boolean AND of multiple boolean *clauses* as follows:

$\mathcal{P} = q_1 \wedge q_2 \wedge q_3 \ldots \wedge q_l$, where, clause $q_i = (a_i, v_i)$ is an attribute-value pair for an attribute a_i, which is true, for an object x, if the value of a_i attribute of object x compares equal to v_i. If the object does not have attribute a_i, it is false. Object x is said to satisfy $\mathcal{P}$ if and only if $\mathcal{P}(x)$ is true for object x. That is, $q_1(x) \wedge q_2(x) \wedge \ldots \wedge q_l(x)$ is true.

In Sponge, queries are rather general, in that, we can pass not only (a, v) pairs but also other forms of predicate clauses. In that, q_i's can be of the forms like (a_i, r_i) where attribute can take a value from a range $r_i = (low, high)$ and $(a_i op v_i)$, where *op* can be any arbitrary binary operator (e.g., $\geq$). Sponge explicitly considers only equal clauses, that is, attribute-value (a_i, v_i) pair clauses in the predicates. These attributes are treated as $hashable$. Attributes that appear in range clauses or in operator clauses will not be hashed, but the resultant objects against the predicate (satisfying the (a, v) part) would be checked whether returned objects satisfy all clauses whatsoever. One assumption is that each query will contain at least one $hashable$ attribute with (a, v) pair.

In Algorithm 2, we see that a user invokes **`searchObject(query)`** procedure whereas Sponge invokes **`search(query)`** and **`retrieveObject(key)`** procedures. Upon receiving a query, Sponge uses hash functions on each (a, v) pairs in the query predicate to generate a hash key for this query. This key designated the location of the node who might store objects satisfying this query. A standard DHT lookup is initiated that finds the location of the requested node and meta information stored at that node is searched to find objects that satisfy the query. Note that the same hashing process is used while fusing objects' meta information across nodes (while inserting objects) and while serving a query. If two cases (insertion and search) are constituent of the same set of attribute-value pairs, they are bound to meet at the same node (unless some nodes failed in between or some joins, which are handled separately).

Sponge can handle network dynamics. In that, if the system does not find any suitable results, i. e., query returns empty, it might indicate that the target node for query might be different from the node where metadata was stored. This can be happened due to some node joining or some node failing. Sponge then initiates a search discarding one of the attributes at random from the original one. This now might map to another node that might contain correct results. This process continues until a threshold amount of time is elapsed or no result is found for current combination of attributes. From this scenario, service to the query can be divided to two parts. Part one serves when the number of attributes traced from query string is less than or equal to the number of attributes that has been used to store an object to a peer node and the other part serves when the number of attributes traced from query string is higher. We have discussed these two types of query as follows.

Let, $|q|$ denotes the attribute list traced from query string and k denotes the maximum combination of attributes that has been used to store meta information to a peer node. Then we find two cases to serve response to any query.

When $|q| \leq k$ **:** In this case, our system can answer exactly provided that the list of attributes is valid.

When $|q| > k$ **:** In this case, our system cannot provide exact answer rather than serve search results taking first k attributes under consideration only.

Search procedure provides a list of keys which is the result of provided query string. If user wants to download and install an app in his/her device, any of those keys will be used to fetch the original app content respectively invoking **`retrieveObject(key)`** procedure.

C. Overhead Allowance

Due to various combination of attributes, a reasonable amount of overhead is introduced throughout the system. We have made this possible in our system to keep it tunable. A system administrator may allow a percentage of original storage as overhead. In this subsection, we present a road map estimation for overhead allowance in the proposed system.

Let the number of objects be N, the number of attributes for each object be n, average size of each object is denoted by s, size of meta data for each object be m and k denotes the allowed combination of attributes in the system. Then meta information of each object will be inserted to $\binom{n}{k}$ peers. The total size of meta data for N objects will be $N\binom{n}{k}m$. As each object has an average size of s, total storage for N objects will be $Ns + N\binom{n}{k}m$. So, system overhead can be calculated as,

$$\begin{aligned} Overhead &= \frac{N\binom{n}{k}m}{Ns} \\ &= \binom{n}{k}\frac{m}{s} \end{aligned}$$

Here, $\frac{m}{s}$ is termed as meta ratio. If we would like to allow 10% overhead in the system having 100 nodes with the parameters value $n = 10$, $m = 100$ byte, $s = 100$ KB, then we will to find a suitable value for k as follows:

$$\begin{aligned} &\binom{n}{k}\frac{m}{s} \leq 0.1 \\ or, &\binom{10}{k}\frac{100}{100000} \leq 0.1 \\ or, &\binom{10}{k} \leq 100 \end{aligned}$$

here, for $k = 1, 2, 3$, the value of $\binom{10}{k}$ will be $10, 45$ and 120 respectively. In this case, a wise choice for k is 2. So we can say that for the give parameters value above, if we would like to allow 10% in the system we must restrict the value of as $k \leq 2$. To this end, it must be ensured that the total number of nodes in the system will be enough to promote $\binom{n}{2}$ peer nodes. A network administrator can allow any required overhead to the system using the equation, $\binom{n}{k} \leq \frac{overhead}{meta\ ratio}$.

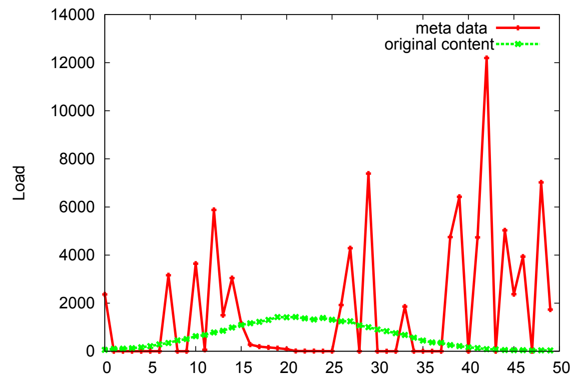

Fig. 2: Load distribution among 50 peer nodes showing meta data and original content distribution to different peers.

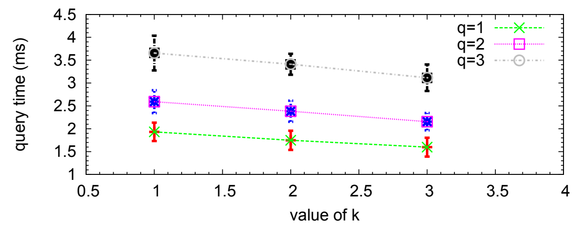

Fig. 3: Average query time for various values of k with standard deviation where query string contains q *Hashable* attributes.

IV. Experimental Results

We have implemented the Sponge system using Python v. 2.7.6 language with standard $kademlia$ package. The experimental PC is configured as Windows 8, Intel Core2 duo 2.2GHz, 2GB RAM. We have conducted the experiments for both multi-attribute insertion and retrieval operation.

We run several type of experiments on our implemented system for different experimental set ups. In our system, all nodes are peer nodes. A peer node may contain both the original objects (data items) and the meta data information about the objects. In Figure 2, we show load distribution to different nodes. For this experimental set up, we use 50 nodes and 10^5 objects. We see that different peer nodes contain different volume of load distribution. However, some peers have no load distribution. This is because of no id generation of that peers after performing hash function on different combination of attributes. We also see that peers who contain the original content follows a power distribution law.

A measurable amount of messages are passed throughout the system during insertion, searching and retrieval of objects. During this message passing, a significant amount of network bandwidth is consumed. So, we keep here the number of message passing as small as possible. In our system, it needs $\sum_{i=1}^{k}\binom{n}{i} + 1$ messages while inserting an object into the system. While searching and retrieval of object from the system, it needs a constant amount of messages except node failure. As it needs some network latency to send messages, in the simulated environment, we have assumed 200 ms delay as round trip.

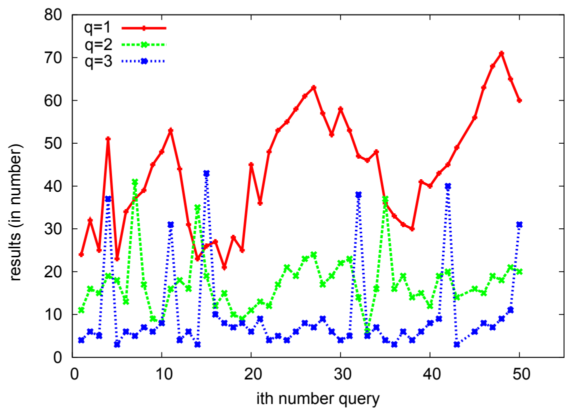

Fig. 4: Result-set size for i^{th} query where query string contains q *Hashable* attributes.

In Figure 3 we show average query time for different values of k ($k = 1, 2, 3$) (excluding network latency). In this Figure, value of q denotes the number of attributes extracted from query string. For this experimental set up, we use 10^2 nodes, 10 types of attributes and $10^3 - 10^4$ objects. Here, we also show the standard deviation for query processing for various values of k. We see that time variation for several query processing is very small. Obviously, our system responds for a single query within fraction of a second.

In Figure 4, we show another experimental results which about the results size of a query. Results contain the number of $keys$ for corresponding objects. In this Figure, we see that when query string contains one attribute ($q = 1$) then our system responds with higher volume of result set rather than other two. Notably, we see that there are some inflations in the results for $q = 2, 3$. This is because of not getting exact answer for the provided attributes. Then system initiates a search discarding an attribute randomly which results in larger volume of output.

We analyze the facilities of storing objects using combinations of different attributes. When objects are stored using single attribute only then a query having more than one attribute is served as an intersection of results for a sequence of queries containing each of those attributes. Simply, this type of service takes long time to respond. In this case, a single query takes 202.3 ms (including network latency) on an average to serve. But when combination of attributes is used to store objects to peer nodes then it takes 201.37 ms (including network latency) on an average to serve. So, it definitely facilitates to serving queries in low latency but this will introduce higher system overhead.

V. Conclusion

In this paper, we have presented a P2P system to show the underlying structure of a distributed app store which gives results for both single attribute and multiple attributes i.e., search string can contain multiple words. Our performance results shows that it performs well to process distributed searching of object(s) from the system in low latency. We have also shown that our system facilitates a network administrator to allow any required overhead. Any system that needs multi attribute queries in time-sensitive, light-weight and efficient manner, will get benefits from our system.

References

[1] https://play.google.com/store

[2] http://en.wikipedia.org/wiki/Google_Play

[3] https://www.apple.com/itunes/

[4] http://store.apple.com/

[5] http://store.handmark.com/

[6] http://apps.opera.com/

[7] http://www.handango.com

[8] http://en.wikipedia.org/wiki/Web_cache

[9] C. L. Giles, K. D. Bollacker and S. Lawrence, *CiteSeer: an automatic citation indexing system*, In Proc. of DL, third ACM conference on Digital libraries, Pages 89-98, New York, 1998.

[10] R. Escriva, B. Wong, and E. G. Sirer, *HyperDex: A Distributed, Searchable Key-Value Store*, In Proc. of ACM SIGCOMM, Helsinki, Finland, August, 2012.

[11] Md Ahsan Arefin, Md Yusuf S Uddin, Indranil Gupta, Klara Nahrstedt. *Q-Tree: a Multi-attribute Based Range Query Solution for Tele-immersive Framework*, ICDCS 2009, Montreal, Canada, June 2009.

[12] A. T. Clements, Dan R. K. Ports, D. R. Karger, *Arpeggio: Metadata Searching and Content Sharing with Chord*, In Proc. of IPTPS, Ithaca, NY, USA, February 2005.

[13] I. Stocia, R. Morris, D. Karger, M. F. Kaashoek, and S. Shenker, *Chord: A scalable peer-to-peer lookup service for internet application*, In Proc. of ACM SIGCOMM, San Deigo, CA, August 2001.

[14] A. Rowstron and P. Druschel, *Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems*, In Proc. of IFIP/ACM ICDSP, 2001.

[15] A. Chrainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram, *P-ring: An efficient and robust p2p range index structure*, In Proc. of SIGMOD, 2007.

[16] Fay Chang et al. *BigTable: A distributed storage system for structure data*, ACM TOCS 26.2 (June 2008), 4:1-4:26.

[17] G. DeCandia et al. *Dynamo: Amazon's highly available key-value store*, In Proc. of SOSP, Stevenson, Washington, USA, October, 2007.

[18] P. Maymounkov, D. Mazires, *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric*, In Proc. of IPTPS, London, UK, 2001.

[19] R. Renesse and K.Binnan, *Astrolabe: A robust and scalable technology for distributed system monitoring, management and data mining*, vol. 21, no. 2, pp. 164-206, May 2003.

[20] P. Yalangandula and M. Dahlin, *A scalable distributed information management system*, In Proc. of ACM SIGCOMM, 2004, pp. 379-390.

[21] J. Liang, S. Ko, I. Gupta and K. Nahrstedt, *Mon: On-demand overlays for distributed system management*, In Proc. of USENIX WORLDS, 2005.

[22] S. Ko, S. Yalagandula, I. Gupta, V. Talwar, D. Milojicic, and S. Iyer, *Moara: Flexible and scalable group-based querying system*, In Proc. of ACM/IFIP/USENIX Middleware, 2008.

[23] O. Gnawali, *A keyword set search system for peer-to-peer networks*, Master's thesis, Massachusetts Institute of Technology, June 2002.

[24] Y. Mao, E. Kohler, and R. T. Morris, *Cache Craftiness for Fast Multicore Key-Value Storage*, In Proc. of EuroSys, Bern, Switzerland, April 2012.

[25] H. Lim, D. Han, D. G. Andersen and M. Kaminsky, *MICA: A Holistic Approach to Fast In-Memory Key-Value Storage*, In Proc. of NSDI, Seattle, WA, USA, April, 2014.

[26] C. Mitchell,Y. Geng and J. Li, *Using one-sided RDMA reads to build a fast, CPU-efficient key-value store*, In Proc. of the 2013 conference on USENIX Annual technical conference June, 2013.

[27] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia and J. Ousterhout, *SLIK: Scalable Low-Latency Indexes for a Key-Value Store*, Under Review, July, 2014. Accessible at https://ramcloud.stanford.edu/wiki/download/attachments/11960661/slik.pdf