Also publ. as: Conference Paper IEEE LCN November 2004

Bringing Efficient Advanced Queries to Distributed Hash Tables

Daniel Bauer Paul Hurley Roman Pletka Marcel Waldvogel
IBM Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland
{dnb,pah,rap,mwl}@zurich.ibm.com

Abstract

Interest in distributed storage is fueled by demand for reliability and resilience combined with ubiquitous availability. Peer-to-peer (P2P) storage networks are known for their decentralized control, self-organization, and adaptation. Advanced searching for documents and resources remains an open problem. The flooding approach favored by some P2P networks is ineffiencient in resource usage, but more scalable and resource-efficient solutions based on Distributed Hash Tables (DHT) lack in query expressiveness and flexibility. In this paper, we address this issue and introduce new efficient, scalable, and completely distributed methods that strive to keep resource consumption by queries and index information as low as possible. We describe how to improve the handling of multiple subqueries combined through boolean set operators. The need for these operators is intensified by applications to go beyond simple exact keyword matches. We discuss, optimize, and analyze appropriate extensions to support range and prefix matching in DHTs.

1. Introduction

Interest in distributed storage is fueled by demand for reliability and resilience combined with ubiquitous availability. Peer-to-peer (P2P) storage networks are favored for their decentralized control, self-organization, and adaptation.

A key function of such a network is the location of resources. Location comes in two distinct flavors: a *lookup* by resource ID or a *query* for resources matching certain properties. For lookup functionality, Distributed Hash Tables (DHT) are generally considered to provide the most efficient solution. However, for general queries, message flooding is still be relied upon, which attains flexibility at the expense of inefficient resource use. We show how to perform many important classes of queries efficiently in DHTs, thereby circumventing the need for flooding.

The many forms DHTs come in include space division [16, 23], interval routing/skip lists [9, 13, 19], or tree-like structures [1, 15, 18, 25]. They all have a common interface, which resembles a hash table, with support for `get`, `put`, and `remove` of elements addressed by a unique, exact key.

1.1. Basic Search Methods

In current highly distributed databases and peer-to-peer storage systems, three basic methods exist to search for a resource. The first directs a search query to a centralized directory. It is evaluated there, the data blocks that correspond to the search query are determined, and the result of the query transmitted back to the requestor. A disadvantage is that there are no alternatives to avoid or compensate for failures, as there is a single point of failure – the centralized directory. There are also scalability issues, as every request will need to be handled centrally. On the positive side, this approach does allow for very powerful and flexible queries.

The second method uses an identifier-only (ID-only) access, which can be carried out with the help of a DHT. Drawbacks include that the ID needs to be known and that no queries other than a request for a resource with a particular ID are possible.

The third method is to use a flooding query. A benefit of this approach is that all kinds of methods of matching queries can be implemented. Unfortunately, flooding is extremely expensive in terms of both network bandwidth and computation at nodes.

1.2. Alternative and Extended Methods

In addition to these three basic methods, alternatives to improve some of their deficiencies exist. For example, additional replicated directories instead of a single centralized directory may be used. However, there

is still a small set of dedicated failure points, and moreover, a high load may arise on the replicas, both from queries and updates.

Another alternative is to use stored or cached queries (employed, for example, in INS/Twine [3]), where IDs are assigned to each expected query using, for example, a DHT. When querying, some variants of the query are tried.

1.3. Contributions

We present *CANDy (Content Addressable Network DirectorY)*, a general framework for querying, in a completely distributed fashion, resources stored in DHT-based systems. It extends the existing work in this area by offering flexible queries while preserving the DHT efficiency.

Proceeding beyond keyword and substring searches, we introduce value matching against distributed databases that consist of ranges or prefixes. In addition, it is shown when and how Bloom filters [4] can (or cannot) help in the search process. Finally, a comparative analysis of these mechanisms with existing work is provided.

The remainder of the paper is structured as follows. The next section describes the design principles, which are then applied to advanced queries such as a set of words, range searches, and longest prefix match in Section 3, and optimized using Bloom filters in Section 4. Section 5 briefly assesses total storage consumption, the number of messages transmitted, and the scalability and resiliency of our approach. Finally, related work is discussed in Section 6 and the paper concluded in Section 7.

2. CANDy Design Principles

DHTs provide a very efficient lookup of resources when the resource identifier (resID) is known. Queries, on the other hand, ask for resources that have certain properties. CANDy's approach is to use a DHT that links properties to resIDs. The idea is to compute a property identifier (propID) for each property of a resource and store the resID at the location of the propID. As resources share properties, an entire set of resIDs are stored under the same propID.

The *index DHT* stores property and index information, while the *resource DHT* stores the actual resources traditionally. Logically, these are separate entities but can be implemented as part of a single DHT infrastructure.

In CANDy, a query is processed as shown in Figure 1. The user agent analyzes the query, identifies the properties involved and resolves the data type of each property. This is done using property descriptors. Each property represents a set of resIDs. The user agent translates the query into a sequence of set operations on resID sets. The user agent sends the query, now in the form of a sequence of set operations, to the storage nodes that store the corresponding resID sets. Each node processes a subquery as the query is handed from node to node. The last node returns the result to the user agent. The result is a (possibly empty) set of resIDs. The user agent then uses these resIDs to accesses one or more resources using a DHT lookup in the resource DHT.

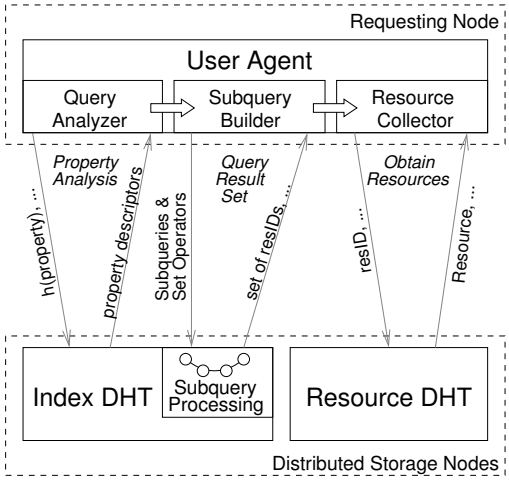

Figure 1. Component interaction in CANDy.

This process will now be described in more detail.

2.1. Properties and Property Descriptors

Properties are name/value pairs of a certain type that describe resources. For example, an image document has a property `title` of type `string` with value "Sunflowers". The propID is computed using a hash-function $h()$ that is applied to both name and value, i.e., propID = h("title=Sunflowers"). The propID is then used for the propID-lookup.

As there are several ways of storing properties in the index DHT, it would not be sufficient to know just the property name. An additional data structure, the property descriptor, is used to store information about property types and how they are stored in the index DHT. For example, the property descriptor for `title` indicates that `title` is of the type `String` and stored as `set-of-words`. For other data types, property descriptors contain more information. In particular, comparable data types can be stored in hierarchical structures that allow for range searches, cf. Section 3. Prop-

erty descriptors are stored in the index DHT, using the name of the property to compute the storage ID. Using the above example, the property descriptor for `title` is stored using the ID $h(\text{“title”})$.

2.2. Subqueries and Set Operations in Index Nodes

The result of a propID-lookup operation in the index DHT is a set of resource IDs. A typical query asks for resources that fulfill several property names or values rather than one single property. This means that several propID lookups have to be combined to fulfill the query. This could be done locally at the requesting node, i.e., all resID sets could be transferred to the requesting node, which then carries out the evaluation. As resID sets may be large, such an approach requires a large amount of network resources. CANDy uses a distributed approach in which the query, in the form of a sequence of set operations, is forwarded from one storage node to another where individual propID lookups are evaluated on the fly. As the query propagates, intermediate results in the form of resID sets are added until the final result is returned to the requesting node.

Hence we propose the following scheme: At the requesting node, CANDy splits the search query into subqueries and a set of operations that links them. This query comprises all information on the resources to be located in the distributed storage nodes. The supported set operations are union '$\cup$', intersection '$\cap$', and set minus '$\setminus$'. The complement '$\overline{A}$' can be obtained from $A \cap \overline{B} = A \setminus B$. Note that the complement operator only makes sense in conjunction with a given base set, where it can be computed using set minus.

A query request includes the subqueries and the set of operators. This information is encoded in a linearized evaluation tree, e.g., using reverse polish notation (RPN) as instructions for a stack machine. The query request is then transmitted sequentially along storage nodes which processes one subquery and links the result provided by the previous node by means of the corresponding set operator. The request is then adapted and the request forwarded to the next node accordingly. Eventually, the last node sends the final result back to the requesting node. The packet is routed between nodes by the underlying DHT.

Consider the example where the keyword query for all resources that do not contain sunflowers and portrait paintings by Vincent Van Gogh could be (“artist=Van”, “artist=Gogh”, not (“title=Sunflowers” or “title=Portraits”)), which can be translated into four subqueries and written as

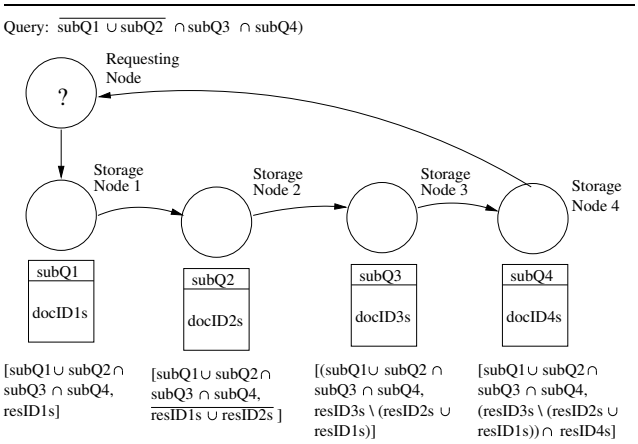

Figure 2. Search example with subqueries.

$\overline{\text{subQ1} \cup \text{subQ2}} \cap \text{subQ3} \cap \text{subQ4}$, where

$$\begin{aligned} \text{subQ1} &= \text{“title=Sunflowers”} \\ \text{subQ2} &= \text{“title=Portrait”} \\ \text{subQ3} &= \text{“artist=Van”} \\ \text{subQ4} &= \text{“artist=Gogh”}. \end{aligned}$$

The subqueries and their set operators are then sent through the network as illustrated in Figure 2 with each storage node performing one subquery. Each subquery returns the resIDs of the resources that contain the corresponding keyword. The resID sets are then combined using the corresponding set operator before the packet is forwarded to the next node. In particular, for storage node 2 it does not make sense to enumerate the complement set $\overline{\text{resIDQ1} \cup \text{resID2s}}$, hence resIDQ1 $\cup$ resID2s is marked as being a complement set instead.

Note that when using an existing DHT algorithm to route the query packet to the next node, $O(\log n)$ intermediate nodes are expected to be traversed until the node that executes the next subquery is reached. These intermediate nodes are omitted from Figure 2.

If the index DHT has a message size limitation, the message may be split into multiple segments, each accompanied by the range of resIDs they represent. This is necessary to allow independent processing of these messages while ensuring correct operation of later set complement or exclusion operations. Messages so split can travel independently and without reassembly to the destination.

A query does not necessarily require the entire set to be transfered to the next node before a set operation can take place. When the sets are all ordered by a common criteria, the set operations can be performed in streaming mode [11].

2.3. Building the Final Result

The resource collector first gathers all resIDs, which might be spread over several subquery result packets, and combines them into the final set of resIDs. From this set, each resource is then found using the resource DHT. Note that, depending on the complexity of the query, the resIDs might come from more than one node. In such a case, the last set operation is carried out by the resource collector before accessing the resources.

3. Complex Searches in Distributed Storage Nodes

In this section we present three different examples that utilize the query architecture based on subqueries introduced above, thereby allowing more complex search queries to be formed, namely queries consisting of a set of words such as property values, range searches and longest-prefix match searches.

3.1. Set-of-words Searches

If a property value is composed of multiple words, then it may be advantageous to use multiple propIDs. For example, an image document with title "Yellow House" is stored under both h("title=Yellow") and h("title=House"). This allows the further decomposition of queries leading to increased distributed processing and it also allows the location of a resource by means of a single word.

3.2. Range Searches

Often, it is desirable to search for value ranges, rather than queries that are exact or contain solely keywords.

Let's consider two possible implementations. The first optimizes node access and uses only one node access to retrieve the set of resIDs for any possible subrange. Storage requirements are high, up to $W \cdot (W+1)/2$ index resources are required, each containing a distinct subset of the resIDs, where W is the size of the value space V; $W = |V|$. The second optimizes storage such that each resID is stored only once. This results in $U - L + 1$ requests for index documents, where L and U are the lower and upper bounds of the search range. These two extremes show that either a lot of memory space or a large number of queries are necessary to perform a range search.

Hence, the amount of storage space required – a static requirement – and the number of nodes in the network to visit or index resources to retrieve – a dynamic requirement – must be optimized. Both should be as small as possible but might have different priorities depending on the application scenario.

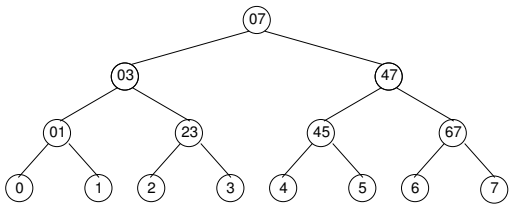

Figure 3. Example range search tree.

A promising method to perform range searches efficiently is to organize the range values in a tree as shown in Figure 3. In the example tree for the bounded value space $V = \{0, 1, \ldots, 7\}$, the values are associated with subranges that are super-ordinated to higher-level subranges. The example shows a four-level tree. The leaves comprise individual values. Here, the first leaf from the left has the value 0, and the first node on the next level up covers the subrange from 0 to 1. The root (top-level node) represents the full value space V. From the explicit tree structure or one defined by an algorithm, values and value ranges can be found. Value ranges use an algorithm that computes the largest common subranges of the search range. The index resource on each node contains all resIDs that match the corresponding value range. In addition, further information on parent and child nodes, the branching factor, the number of levels and query encoding is stored in the property descriptor of the corresponding propID.

For example, when searching in the range $\{2, \ldots, 6\}$, three subqueries are generated: The first for the content of index document 23, the second for index document 45 and the third for index document 6. Without a tree structure, the same search range results in five single queries (one for each value). Note that this saving effect is significantly more pronounced for larger value spaces with larger trees.

When the value space reflects a continuous range each node represents a bounded continuous subrange. In both discrete and continuous ranges the granularity of the tree structure is chosen such that the resID sets in the leaves are kept small (e.g., checking only the first p letters in a string search). In general, the storage requirements of the tree structure and the number of queries required is $\log_k W$.

Next, we analyze the worst-case performance for a fully populated and balanced tree. The number of levels of the tree structure is, $\ell = 1 + \lceil \log_k W \rceil$. Under the assumption that union is the sole operation supported by the DHT, the necessary worst-case storage factor S for a branching factor k is $S = \ell$; i.e. each

resID is stored at each level of the tree. The number of queries Q_u in the worst case corresponds to

$$Q_u = \begin{cases} 2(k-1)(\ell-2)+k-2, & \ell > 2 \quad \text{and} \\ k-1, & 1 \le \ell \le 2 \quad . \end{cases}$$

The proof which involves induction over ℓ is omitted. From Figure 3 it can be seen that $\{1,\ldots,6\}$ is the worst-case range for union, which can be obtained by unifying the four subqueries sent to the nodes using $\mathcal{N}_1 \cup \mathcal{N}_{23} \cup \mathcal{N}_{45} \cup \mathcal{N}_6$. The worst-case range for union is given by combining the maximum number of nodes on each level which cannot be replaced by a higher-level node and equals $2(k-1)$ (left and right part of the tree) for each level up to level $l-2$ and $k-2$ for the middle part of the tree (i.e., when $k > 2$).

If the operation set minus ($A \backslash B = A \cap \overline{B}$) is also supported, then the number of worst-case search queries for this combined case can be further reduced to

$$Q_c = \begin{cases} (k-1)(\ell-1), & W > 1 \quad \text{and} \\ 1 & W = 1 \quad , \end{cases}$$

based on the observation that any query consisting of more than $k/2$ nodes can be converted to subtracting the remaining nodes from the parent. The necessary worst-case storage is not affected by this operation. As an example, the range $\{1,\ldots 6\}$ can then be expressed as $\mathcal{N}_{07} \setminus \mathcal{N}_0 \setminus \mathcal{N}_7$ resulting in only three subqueries.

3.3. Longest Prefix Match

Another lookup method that has gained popularity is longest prefix match. It is commonly used to determine efficiently into which of several categories a particular identifier (number, string, ...) falls.

Tries have traditionally been used to perform longest prefix matches [7]. Applying them to DHTs can be done in a similar manner to the range searches described above. A property of the tries, that nodes near the root are more frequently accessed, is useful within a single system, as performance is improved through caching effects. In a distributed system, however, where many independent nodes may perform queries at the same time, this is prone to storage node overload.

When the total access time to retrieve information is significant but the per-node access time is high (such as in DHTs), large degrees of nodes become necessary [21]. As a result, memory requirements skyrocket, and update performance deteriorates. A better solution is thus needed.

Binary search on prefix lengths [22, 24] seems to be the ideal candidate, as it is already based on hashing. To use it, all prefixes are simply stored in the DHT. Binary search requires a three-way comparison to decide whether the final solution has already been found or whether the search needs to go on: if so, in which direction. Hash lookups only provide hit or miss answers. To ensure that all entries can be found, some *marker nodes* need to be added to guide the search towards longer prefixes if a natural higher-level prefix does not already exist. When applying the techniques described in [22, 24], the search time is bounded by the logarithm of the search depth.

4. Optimization using Bloom Filters

4.1. Bloom Filter Arithmetic

Bloom filters [4] have proven extremely useful for their extremely compact representations of set membership, which results in small amounts of data to be transmitted [5]. The compactness comes at the expense of possible false positives, the rate of which can be tuned by changing the size (and thus the density) of the filter. Another drawback is that the set can no longer be enumerated, at least not if the domain of the set is sizable. The Bloom filter is limited to set membership queries.

A Bloom filter is a bitmap that represents the union of bitmaps of the individual entries, in which an entry is represented by a few bits set to one, whose position have been selected by a small number of hash functions. Abstractly, a Bloom filter representation $\mathcal{B}(X)$ of X can be viewed as a non-enumerable version of X with some false positives $\varepsilon(X)$, $\mathcal{B}(X) = X \cup \varepsilon(X)$.[1] This results in the following operations on Bloom filters. All binary operations are commutative and associative.

Converting from set to Bloom filter: A Bloom filter can be created from a set: $X \to \mathcal{B}(X)$.

Intersection of Bloom filters: Two Bloom filters of the same size and using the same hash functions can be intersected by logical AND of their bitmaps: $\mathcal{B}(X \cap Y) = \mathcal{B}(X) \cap \mathcal{B}(Y)$.

Union of Bloom filters: Two Bloom filters of the same size and using the same hash functions can be united by logical OR of their bitmaps: $\mathcal{B}(X \cup Y) = \mathcal{B}(X) \cup \mathcal{B}(Y)$.

Bloom filters and enumerable sets: Probing a Bloom filter for presence of each element of an enumerable set leads to a set with false positives:

1 For ease of explanation, we neglect that the (infinite) set of false positives also depends on the size of the Bloom filter and its constituent hash functions.

$\mathcal{N}_? \rightsquigarrow \mathcal{N}_A$: What is $A \cap B$?	Query
$\mathcal{N}_A \rightsquigarrow \mathcal{N}_B$: $\mathcal{B}(A)$	Bloom filter of A
$\mathcal{N}_B \rightsquigarrow \mathcal{N}_A$: $\mathcal{B}(A) \cap B$	Approximate result
$\mathcal{N}_A \rightsquigarrow \mathcal{N}_?$: $A \cap B = \mathcal{B}(A) \cap B \cap A$	Final result

Figure 4. Distributed intersection

$\mathcal{N}_? \rightsquigarrow \mathcal{N}_A$: What is $A \cap \cdots \cap Z$?
$\mathcal{N}_A \rightsquigarrow \mathcal{N}_B$: $\mathcal{B}(A)$
$\mathcal{N}_B \rightsquigarrow \mathcal{N}_C$: $\mathcal{B}(\mathcal{B}(A) \cap B) = \mathcal{B}(A \cap B)$
$\vdots$
$\mathcal{N}_Y \rightsquigarrow \mathcal{N}_Z$: $\mathcal{B}(\mathcal{B}(A \cap \cdots \cap X) \cap Y) = \mathcal{B}(A \cap B \cap \cdots \cap Y)$
$\mathcal{N}_Z \rightsquigarrow \mathcal{N}_Y$: $\mathcal{B}(A \cap B \cap \cdots \cap Y) \cap Z$
$\mathcal{N}_Y \rightsquigarrow \mathcal{N}_X$: $\mathcal{B}(A \cap B \cap \cdots \cap Y) \cap Z \cap Y$
$\vdots$
$\mathcal{N}_B \rightsquigarrow \mathcal{N}_A$: $\mathcal{B}(A \cap B \cap \cdots \cap Y) \cap Z \cap Y \cap \cdots \cap B$
$\mathcal{N}_A \rightsquigarrow \mathcal{N}_?$: $\mathcal{B}(A \cap B \cap \cdots \cap Y) \cap Z \cap Y \cap \cdots \cap B \cap A$

Figure 5. Distributed multi-set intersection

$(X \cap \varepsilon(X)) \cap Y = \mathcal{B}(X) \cap Y$. Obviously, intersecting a Bloom filter of a set with that set will return the original set: $\mathcal{B}(X) \cap X = X$.

4.2. Intersection Operation

Reynolds and Vahdat [17] reduced the message size required for the execution of keyword intersection queries through the addition of Bloom filters.

In the simplest case, where just two sets corresponding to keywords have to be intersected, A and B, stored on nodes $\mathcal{N}_A$ and $\mathcal{N}_B$, the process is shown in Figure 4. Note that $\mathcal{B}(X)$ represents the Bloom filter representing set X and the requesting node is known as node $\mathcal{N}_?$. Reuse of the previous message is indicated by a light gray background.

Given the above set of operations, we can see that $\mathcal{B}(\mathcal{B}(A) \cap B) \cap A = A \cap B$. The advantage is that the sets A and B are never transmitted, only a compact representation $\mathcal{B}(A)$ and a close approximation of the final result. This does not help much if A and B share many elements, but significantly improves the case when A and B are large and have few or no elements in common.

The generalization for intersection among multiple sets is shown in Figure 5. Instead of having a forward path with Bloom filters and a reverse path with approximate sets, the second path can also be performed in forward direction, from Z over A to Y, or in any other order.

$\mathcal{N}_? \rightsquigarrow \mathcal{N}_A$: What is $A \setminus B$?
$\mathcal{N}_A \rightsquigarrow \mathcal{N}_B$: $\mathcal{B}(A)$
$\mathcal{N}_B \rightsquigarrow \mathcal{N}_A$: $\mathcal{B}(A) \cap B$
$\mathcal{N}_A \rightsquigarrow \mathcal{N}_?$: $A \setminus B = A \setminus \mathcal{B}(A) \cap B$

Figure 6. Distributed exclusion

4.3. Exclusion Operation

Another common operation is exclusion ("and not" or "set minus"). This operation seems impossible, as Bloom filters cannot be complemented nor do they support the elimination of elements by clearing some bits in the filter, as this would probably lead to the eliminatation of some other (real) elements of the filter.

Figure 6 presents our solution to the binary exclusion problem. The algorithm is based on $X \setminus Y = X \cap \overline{Y} = X \cap \overline{X \cap Y}$. It is a shortcut calculation of intersection (lines 2 and 3), but the combination in step 4 is different. A generalization where $(A \cap \cdots \cap M) \setminus (N \cap \cdots \cap Z)$ can be calculated as shown in Figure 7.

Path separation is not needed if there is only a single set to the left of the exclusion operator. Instead, the mechanism described in Figure 5 can be used almost verbatim, with only the last intersection symbol in the result message changed from intersection to exclusion.

It could be argued that the process can be simplified by calculating the intersection among the base set, $A \cdots M$, and the excluded set, $n \cdots z$, separately, and performing the exclusion only at the very end. Although this would lead to the correct result, the amount of data transmitted would, in general, be unacceptable. The intersection among $N \cdots Z$ could be huge compared to the base set and the final result. The additional inclusion of the Bloom filters for $A \cdots M$ into the calculation of the excluded set prevents the protocol from transmitting much more data than is part of the base set, and thus is not relevant to determining the result set.

If the intersection between the base set and the excluded set is large in comparison to the result set, it would be useful to avoid transmitting it as elements, but keep it only as a Bloom filter representation. Unfortunately, the solution above is the best known approach and, because of the false positives involved, these improvements do not seem feasible.

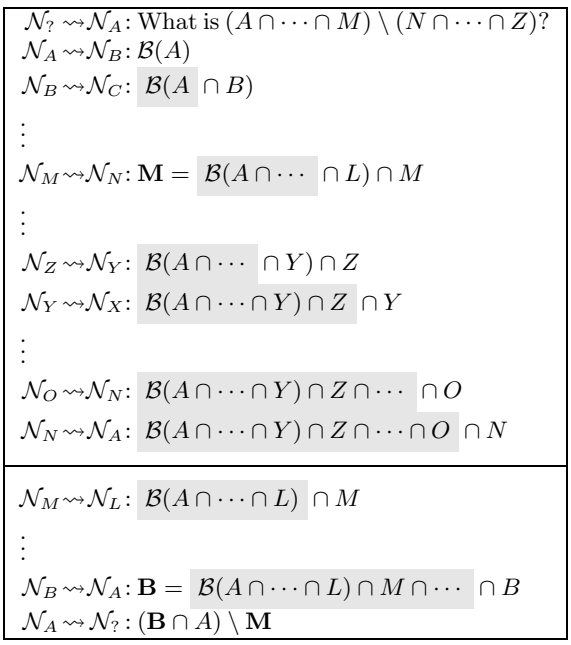

Figure 7. Distributed multi-set exclusion supported by Bloom filters

4.4. Union Operation

The third major operator used in set operation is union (inclusion, conjunction). Distributed operation of the union operation requires each element of the result set to be transmitted over the network at least once. The straightforward implementation of the (binary) union operation described earlier consists of individual transmission of the contributory sets to the next hop in the course of query processing. The elements shared by the two contributory sets are transmitted to the destination twice. If the sets are identical, which is the worst case, the data transmission doubles compared to the optimum transmission.[2]

Unfortunately, it does not seem possible to do better than the straightforward implementation, because the shared elements need to be communicated between the storage nodes to prevent duplicate delivery. This in turn involves transmitting the shared elements over the network twice: once between the storage nodes and once from the storage nodes to the next hop.

Even though this means that the total transmission size cannot be improved, using Bloom filters to calculate the shared set (=intersection) and then preventing duplicate delivery to the next hop can be advantageous, especially when the recipient of this (potentially intermediate) result has limited bandwidth.

To process unions as part of Bloom-filter-optimized queries (e.g. as explained in Figure 7), any term can be considered to be constructed out of unions as a disjunctive normal form. The conjunctions will be evaluated as a lower-level subquery and their result used for the higher-level query sequence. This is applicable to unions in Bloom filter form as well as in an element listing.

5. Assessment

What follows is a qualitatively assessment of the performance of CANDy's compared with a solution based on a central directory and an approach that uses flooding.

5.1. Storage Considerations

The flooding approach uses a set of equal nodes or peers, in which each node stores a subset of the resources together with the corresponding properties. As the resources are stored together with the properties on the same node, no refIDs are required. In terms of storage, the flooding approach is optimal.

The central directory solution uses a central server to store property information and to link property values to resIDs. As all property information is available at the central server, resIDs do not have to be replicated for each matching property. The total storage required for this solution is then given by the storage requirements of the resources themselves and the sum of all stored property values and one resID for each resource. In contrast to the flooding approach, storage space for the resIDs has to be provided. The resIDs are, however, very small, and for all practical purposes the additional storage can be neglected.

CANDy distributes the property information and the associated resIDs across the index nodes. Even though property information is mapped to propIDs by the user agent, index nodes still need to store the complete property information in order to resolve hash collisions. The distributed nature of CANDy requires that an individual resID is replicated multiple times, once for each property that it matches and additional times to allow substring matching and range searches. As the number of properties of a resource is limited, an individual resID will typically be replicated less than 100 times. The storage overhead is small compared to the optimal solution, as an individual resID typically consists of only a few bytes.

2 When the two sets are disjoint, obviously no improvements can be made, as each element will already be transmitted only once.

5.2. Number of Message Transmissions

The flooding approach needs to distribute the query to all nodes to find all matching documents. The performance of flooding depends on the topology, but in order to reach N nodes at least N messages have to be sent. In practice, flooding is much less efficient as an individual node will receive the same message multiple times. Clearly, flooding does not provide a scalable solution.

The central directory solution requires four message transmissions. The query is sent to the directory server, which answers with the set of matching resIDs. An ID is looked up and the resource is accessed. In terms of messaging overhead, the centralized directory approach is close to optimal.

CANDy's messaging overhead is determined by the complexity of the query. In particular, the sequence of set operations also determines the number of message transmissions. As typical queries involve just a very small number of properties, the messaging overhead is also very small, on the order of a few messages.

5.3. Scalability and Resiliency

The flooding approach provides excellent resiliency against node and network failure. As long as a resource is reachable in the network, the flooding approach will find it. The lack of scalability due to excessive generation of network traffic makes the flooding approach unusable for even moderately sized networks.

The central directory approach is very efficient, but lacks both scalability and resiliency. The central server is a bottleneck that can get overloaded by a large number of queries. A failure of the central server has fatal consequences for the whole system.

While CANDy is moderately less efficient than the central directory approach, it provides a scalable and more resilient solution. Due to its distributed approach, it spreads the load across several nodes without requiring excessive network resources. It is resilient in that an index-node failure affects the precision of the result, but not the overall functionality.

6. Related Work

Some recent proposals blur the boundary between structured (i.e., DHTs) and unstructured (Gnutella-style) P2P networks. Cohen et al. [6], for example, take advantage of shared interests to try to attain good results with limited flooding while pSearch [20] brings limited flooding to a modified DHT in order to get soft queries. The remainder of the section, though, will focus on papers sharing our goal of bringing advanced queries to essentially unmodified DHTs.

INS/Twine [3] is an intentional resource discovery system [2], where resources are described by attribute/value trees (AVtree), with the attributes organized orthogonally and hierarchically. Each AVtree is stored and retrieved under a key generated by hashing the tree. While INS/Twine solves the 'exact-match' problem, it produces a potentially large number of keys and consequently stores the same resource description on a large number of different nodes. Set operations, restricted to intersection, are implicit in the construction of the query AVtree.

Felber et al. [8] extend upon the INS/Twine concept by overlaying the DHT with a rooted directed acyclic graph (DAG). The edges of this DAG represent queries that can be used to refine the result set. The actual query process becomes a tight interaction between the requesting node and the DAG.

Approximate range searches are supported by Gupta et al. [10]. When a query for range $[s, e]$ is made, they try to locate a range $[s-\varepsilon, e+\varepsilon']$ that has been stored in the DHT, which minimizes $\varepsilon, \varepsilon'$. To make this fuzzy match, they use distance-sensitive hash functions. The result needs to be postprocessed to evict out-of-range matches.

Substring searches have been described in [12] through the intersection of the sets indexed by the n-grams of the string. Similar to Bloom filters, these are subject to false positives, which then need to be weeded out by inspecting the actual objects.

A completely different approach is taken by PIER [14]. Instead of distributing the query processing, they implement a fully functional relational database on top of a DHT. Their trick is to disperse the database objects into the DHT, essentially treating the DHT as a distributed disk.

7. Conclusions

A major challenge in the use of distributed storage is the location of resources with a flexible and expressive query language, but without reliance on single points of failure or requiring the waste of network or storage solutions.

We have presented CANDy, a solution to the problem for expressive distributed searching that is (i) fully distributed, (ii) efficient and scalable in its use of computation, storage, and communications resources, (iii) modular and flexible, and (iv) is able to take advantage of any underlying DHT technology.

CANDy provides the full range of set operations applicable in this context. These operations are available both to the querying agent and used internally to extend exact matches to range and prefix searches. We also extended the use of Bloom filters for handling exclusion and showed that a further extension to union operations is not reasonable.

Acknowledgments

We would like to thank Jan van Lunteren and Patrick Droz for many helpful discussions.

References

[1] K. Aberer, M. Hauswirth, M. Punceva, and R. Schmidt. Improving data access in P2P systems. *IEEE Internet Computing*, 6(1), Jan./Feb. 2002.

[2] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Symposium on Operating Systems Principles*, pages 186–201, 1999.

[3] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Pervasive 2002 - International Conference on Pervasive Computing*, Aug. 2002.

[4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[5] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Proceedings of the 40th Annual Allerton Conference on Communications, Control, and Computing*, pages 636–646, 2002.

[6] E. Cohen, A. Fiat, and H. Kaplan. A case for associative peer-to-peer overlays. volume 33, Jan. 2003. Proceedings of ACM HotNets-I (October 2002).

[7] W. Doeringer, G. Karjoth, and M. Nassehi. Routing on longest matching prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, Feb. 1996.

[8] P. A. Felber, E. W. Biersack, L. Garcés-Erice, K. W. Ross, and G. Urvoy-Keller. Data indexing and querying in DHT peer-to-peer networks. In *Proceedings of ICDCS 2004*, Tokyo, Japan, 2004.

[9] G. N. Frederickson. Searching intervals and compact routing tables. *Algorithmica*, 15(5):448–466, May 1996.

[10] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, California, USA, Jan. 2003.

[11] R. F. Haddleton. *Parallel Set Operations in Complex Object-Oriented Queries*. PhD thesis, University of Virginia, Jan. 1998.

[12] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of IPTPS 2004*, 2004.

[13] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS '03)*, Mar. 2003.

[14] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, Berlin, Germany, Sept. 2003.

[15] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM*, Sept. 2001.

[17] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *Proceedings of Middleware 2003*, June 2003.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, Nov. 2001.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM SIGCOMM 2001*, pages 149–160, San Diego, CA, USA, Aug. 2001.

[20] C. Tang, Z. Xu, and M. Mahalingam. pSearch: Information retrieval in structured overlays. volume 33, Jan. 2003. Proceedings of ACM HotNets-I (October 2002).

[21] H. H.-Y. Tzeng and T. Przygienda. On fast address-lookup algorithms. *IEEE Journal on Selected Areas in Communications*, 17(6):1067–1082, June 1999.

[22] M. Waldvogel. Multi-dimensional prefix matching using line search. In *Proceedings of IEEE Local Computer Networks*, pages 200–207, Tampa, FL, USA, Nov. 2000.

[23] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. *ACM Computer Communications Review*, 33(1):101–106, Jan. 2003. Proceedings of ACM HotNets-I (October 2002).

[24] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high-speed prefix matching. *Transaction on Computer Systems*, 19(4):440–482, Nov. 2001.

[25] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California, Berkeley, April 2001.