Index Recommendation Tool for Optimized Information Discovery over Distributed Hash Tables

Faraz Memon, Frank Dürr, Kurt Rothermel
IPVS – Distributed Systems Department, Universität Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
Email: {faraz.memon, frank.duerr, kurt.rothermel}@ipvs.uni-stuttgart.de

***Abstract*—Peer-to-peer (P2P) networks allow for efficient information discovery in large-scale distributed systems. Although point queries are well supported by current P2P systems – in particular systems based on distributed hash tables (DHTs) –, providing efficient support for more complex queries remains a challenge. Our research focuses on the efficient support for multi-attribute range (MAR) queries over DHT-based information discovery systems. Traditionally, the support for MAR queries over DHTs has been provided either by creating an individual index for each data attribute or by creating a single index using the combination of all data attributes. In contrast to these approaches, we propose to create a set of indices over selected attribute combinations. In order to limit the overhead induced by index maintenance, the total number of created indices has to be limited. Thus, the resulting problem is to create a limited number of indices such that the overall system performance is optimal for MAR queries. In this paper, we propose an index recommendation tool that implements heuristic solutions to this NP-hard problem. Our evaluations show that these heuristics lead to a close-to-optimal system performance for MAR queries.**

***Index Terms*—Distributed networks, Indexing methods, Distributed data structures, Hash-table representations**

I. Introduction

DHT-based information discovery systems started out with the support for only point queries. However, the need for MAR queries in the P2P application areas such as resource discovery in grid computing [1], P2P video streaming [2], and spatial information discovery [3], led to their extension. The support for MAR queries has been provided by realizing a layer of indexing mechanisms over DHTs.

Primarily, two indexing approaches have been used to leverage DHTs for enabling MAR queries. The first approach indexes the value range of each data attribute individually [4], [5], [6], [7]. MAR queries are resolved by performing multiple single-attribute range queries and then the results are filtered at the query initiator. This approach induces a large network load and therefore does not scale. The second approach indexes the combination of all data attributes [8], [9], [10]. This approach performs well if the queries include ranges over all data attributes (which is generally not the case for P2P information discovery systems). Data attributes that do not appear in queries are considered to be wildcards, and the performance of this approach deteriorates with increasing number of wildcards in queries [9], [11], [10].

Recently, we presented the Optimized Information Discovery (OID) system [11], which introduces a third type of indexing approach for extending DHTs to provide the support for MAR queries. The OID system creates a layer of multiple multi-attribute indices over a DHT. A MAR query is resolved by estimating the performance of the query over each index, and then selecting the one with the best estimate. Although our approach outperforms the previously proposed approaches for extending DHTs [11] in terms of query overhead, the criterion for defining the initial set of indices is based on a simple heuristic. The OID system installs a user-defined number of indices for the most popular queries in the system. This criterion achieves system-wide optimal performance for applications where few popular queries make up for the largest portion of the total queries. However, for P2P applications with few popular queries and a large number of unpopular queries, this criterion would produce a set of sub-optimal indices.

Finding an optimal set of indices, irrespective of a particular type of query popularity distribution, is an NP-hard [12] problem, as the number of index possibilities grows exponentially with the number of data attributes. Therefore, a heuristic-based solution that produces a close-to-optimal set of indices, is highly desirable. In this paper, we present a tool that provides index recommendations for DHT-based information discovery systems. Given a limit for the maximum number of indices and a set of MAR queries that have been previously monitored in the system (*workload*), our tool recommends a set of indices that produces close-to-optimal performance for the workload within the given limit.

The index recommendation tool presented in this paper consists of several index recommendation algorithms. Each algorithm works by creating a set of candidate indices using the unique attribute combinations in the workload queries. The set of candidate indices is usually larger than the user-defined limit for the maximum number of indices. Therefore, the size of this set is successively reduced either by merging some elements or by selecting some while discarding others. Our evaluations show that in the best case, a set of indices recommended by our tool is only 1.5% worse than the optimal set of indices in terms of the overhead.

The rest of the paper is organized as follows: in Section II we give an overview of the related work. System architecture along with the role of the index recommendation tool is discussed in Section III. In Section IV we describe the cost estimation technique for MAR queries. The index recommendation tool is introduced in Section V along with several

To appear in proceedings of 35th Intl. Conf. on Local Computer Networks (LCN'10), pages 1-8, Denver, CO, USA, October 2010.

index recommendation algorithms. In Section VI we present the evaluation results. Finally, in Section VII we conclude the paper with an overview of our future research direction.

II. Related Work

To the best of our knowledge, there is no index recommendation tool available for DHT-based information discovery systems. Nonetheless, index tuning/recommendation techniques have been widely studied in the area of database management systems (DBMS).

There is a fundamental difference between the index recommendation tools for databases and the tool presented in this paper. The index recommendation tools for databases rely on the *SQL Query Optimizer* for evaluating the cost of a query over an index [13], [14], [15], [16], [17]. The estimated cost of a query given by the *SQL Query Optimizer* is highly accurate, because the query optimizer uses a central repository known as the *Data Dictionary*, containing statistical information about the data. Gathering such statistical information in a large, distributed and dynamic P2P network would result in a high overhead. Therefore, our approach only uses the local structural information of the indices to estimate the query cost.

The index recommendation tools designed by Bruno et al. [13], Chaudhuri et al. [14], [15], and Valentin et al. [16] carry out the recommendation process in two steps. In the first step, the execution of the workload queries is simulated. During the simulation process, the indices that appear in the *Query Execution Plans* of the query optimizer are collected. This collection represents the initial set of useful indices. The index recommendation tool discussed in this paper also provides the recommendations in two steps. However, our tool obtains the initial set of indices using the unique attribute combinations that appear in the workload queries (c.f. Sec. V-B).

During the second step of the index recommendation process, the recommendation tools for databases refine the initial set of indices using certain heuristics. The goal in general is to obtain a set of indices that produces the least cost for the queries in the workload.

Given a user-defined limit o for the maximum number of indices, the heuristic presented in [14] generates all possible m-sized ($m < o$) subsets of the initial set of indices. The subset with the least cost for the workload is then chosen as a seed for further processing. Next, an index from the initial set is continuously added to the seed until the size of seed is equal to o. It is unclear, how the value for m is chosen, since the algorithm has prohibitively high execution time if m is large, and sub-optimal recommendations if m is small.

In [15], the heuristic introduced by Chaudhuri et al. reduces the size of the initial set of indices by merging pairs of indices as long as a cost constraint is not violated. Unlike the cost-based merge algorithm (c.f. Sec. V-B1) presented in this paper that tries to minimize the workload cost given a limit o for the maximum number of indices, the algorithm in [15] minimizes the storage, given a cost constraint. Therefore, it is possible that the final set of recommendations is larger than o.

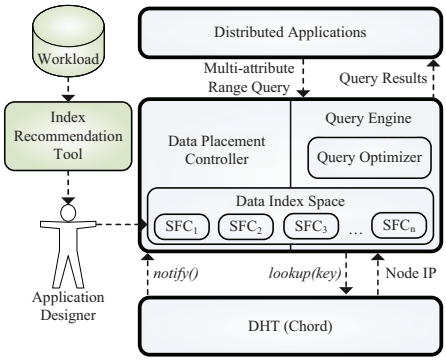

Fig. 1. OID System Architecture

The algorithm introduced by Bruno et al. in [13] produces a set of recommendations by applying different transformations (merging, prefixing, deletion) on pairs of indices in the initial set. The algorithm continues as long as a certain time constraint is not violated. It is not clear, how long the algorithm should be executed so that it produces an o-sized set of recommendations.

Valentine et al. introduced a heuristic in [16] based on a variation of the solution to the Knapsack problem. Their algorithm assigns a cost-to-benefit ratio to each index in the initial set of indices and then selects the indices with the highest ratio as long as a storage constraint is not violated. The algorithm then randomly swaps some indices from the set of selected indices with some indices in the initial set to try another variation of the solution. The swap and selection process continues as long as a time constraint is not violated. Although, the swap and selection technique could lead to a better solution than the initial one, it could also lead to a worse solution.

III. System Architecture

The index recommendation tool presented in this paper provides recommendations for DHT-based information discovery systems with a 3-layer architecture. The top layer consists of distributed applications that require support for MAR queries. The middle layer consists of several multi-attribute indices used for indexing the data and for resolving the queries. The bottom layer is the DHT layer that is used as a distributed lookup service. Fig. 1 shows the architecture of the OID system [11], implementing this 3-layer architecture.

The data index space in the OID architecture is composed of several space-filling curve (SFC)-based multi-attribute indices defined by the designer of the distributed application. The data placement controller uses these indices to assign identifiers to data objects. Each data object is then routed to the peer that is responsible for it. The OID query engine processes a query by estimating the cost of the query on each SFC-based index. The index with the least estimated cost is then used for actual resolution. Additional details of the OID system architecture can be found in our previous work [11].

The index recommendation tool discussed below provides assistance to the designer of the distributed application in order

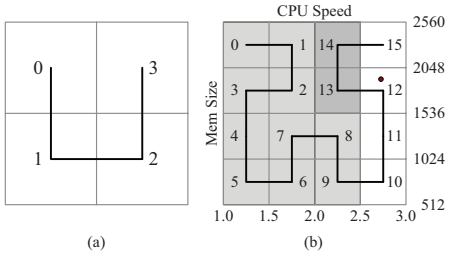

Fig. 2. 1^{st} and 2^{nd} order Hilbert SFCs

to define useful multi-attribute indices for indexing the data. The tool takes a limit for the maximum number of indices and a workload of MAR queries as input. It then recommends a set of indices (SFC-based indices in case of the OID system) that provides close-to-optimal performance for the workload within the given limit. This limit represents a trade-off between increased performance and index maintenance overhead.

We assume that an application-specific workload is available to the designer of the distributed application. Such a workload could be obtained either by analyzing the querying trends of the application domain, or by monitoring queries in an already existing information discovery system. A discussion regarding the approaches for collecting the query workload is beyond the scope of this paper.

IV. Query Cost Estimation

One of the main objectives of the index recommendation tool is to minimize the cost of the workload queries in a DHT network. In particular, we want to minimize the number of peers that are evaluated in order to resolve a query using an index. Given the distributed nature of P2P systems, it is extremely difficult to anticipate the exact number of peers responsible for the resolution of a query. However, a cost function can be defined that allows the index recommendation tool to compare different indices in order to determine the one that, with a high probability, results in the least number of peers being evaluated. In this section, we present such a cost function for the OID system based on the properties of the Hilbert SFC [18].

The OID system uses Hilbert SFCs for indexing the data objects. In the context of the OID system, Hilbert SFC is defined as a continuous function $h : (a_1, a_2, \ldots, a_d) \mapsto x \in \mathbb{N}$, where $(a_1, a_2, \ldots, a_d)$ is a point in a d-dimensional euclidean space and $\mathbb{N}$ is the set of natural numbers. The process of Hilbert SFC construction divides a d-dimensional euclidean space into $2^{k \cdot d}$ sub-cubes, called zones. A line then passes through each of the zones imposing an order on them. The result is a k^{th} order SFC, where k, known as the approximation level, determines the granularity of the space sub-division (c.f. Fig. 2).

If a data object is viewed as a point in a multi-dimensional attribute space, then it can be indexed using a Hilbert SFC. For example, a data object defined as (`CPU Speed` $= 2.7\ GHz$, `Mem Size` $= 1792\ MB$) receives an identifier of 12 on the SFC shown in Fig. 2 (b). Similarly, a MAR query defined as "(`CPU Speed` $>= 1.3$) $\wedge$ (`CPU Speed` $<= 2.3$) $\wedge$ (`Mem Size` $>= 640$) $\wedge$ (`Mem Size` $<= 2304$)" maps to two clusters on the same SFC. A cluster is defined as a list of continuous queried zones. Zones $(0-9)$ and $(13-14)$ are the two clusters in this example (c.f. Fig. 2(b)).

The OID system uniformly distributes the identifier space of a SFC, $[0, 2^{k \cdot d})$, over the DHT [11]. Therefore, with a high probability, the number of peers responsible for query resolution increases with the number of queried zones and clusters. Hence, the index recommendation tool can determine an index that results in the least number of peers being evaluated by calculating the number of zones and clusters for a given query over each SFC-based index. We propose the following cost function for the cost evaluation of each index:

$$cost(q) = \frac{z}{2^{k \cdot d}} \cdot c \tag{1}$$

where z is the total number of queried zones, d is the number of index dimensions, k is the index approximation level and c is the total number of queried clusters. The fraction, $z/2^{k \cdot d}$ (zone fraction), denotes the queried proportion of the SFC.

Calculating the total number of clusters requires calculation of each zone identifier for each SFC-based index. Performing such a calculation could take a significant amount of time. Therefore, the actual number of clusters in Equ. (1) is replaced by the estimated number of clusters, given by d/d_{qm}:

$$cost(q) = \frac{z}{2^{k \cdot d}} \cdot \frac{d}{d_{qm}} \tag{2}$$

where d is the number of dimensions of the index and d_{qm} is number of matching dimensions of the query with the dimensions of the index.

The total number of queried clusters is estimated based on the observation that the number of queried clusters increases with increasing index dimensions and decreases with each matching dimension of the query with the index. Although the exact quantitative increase or decrease in the number of clusters is not known, our simulations show that a proportional relationship is sufficient for a qualitative comparison of different indices (c.f. Sec. VI-B).

V. Index Recommendation Tool

In this section, we discuss the index recommendation algorithms implemented by the index recommendation tool. Given a workload W and a user-defined limit o for the maximum number of indices, each index recommendation algorithm searches for the o most efficient indices for the workload.

The index recommendation algorithms discussed below are independent of a particular type of cost function or indexing technique. Any type of DHT-based indexing scheme and a cost function that estimates the cost of a query, could be used in these algorithms. However, we utilize the SFC-based indices and the cost function discussed in Sec. IV for evaluating these algorithms. The following table defines the symbols used in the discussion below.

Symbols	Definition
$A = \{a_1, a_2, a_3, \ldots, a_n\}$	Set of attributes used in queries
o	User-defined limit for maximum number of recommended indices
$W = \{q_1, q_2, q_3, \ldots, q_m\}$	Set of representative queries. Also known as the workload
$minCost(q, I)$	Returns the minimum cost of a query q on a set of indices I
$attributeCombination(q)$	Returns the combination of attributes used in a query q
$createSFCIndex(x)$	Returns an SFC-based index created using the attribute combination x

A. Naïve Index Recommendation

The most naïve way of determining the optimal set of indices for a workload W is to enumerate all o-sized combinations of the power-set of A (c.f. Algo. 1: line 2), and then choose the combination with the least total cost for the queries in W (c.f. Algo. 1: line 4-9). We call this algorithm the naïve index recommendation algorithm.

Algorithm 1 Naïve Index Recommendation Algorithm

1: $P = Pow(A) - \{\}$;
2: $C = \{c_1, c_2, c_3, \ldots, c_l\} : c_i \subseteq P, |c_i| = o, c_i \neq c_j \ \forall i \neq j$;
3: $minTotalCost = +\infty$; $totalCost = 0$; $R = I = \emptyset$;
4: **for all** c_i in C **do**
5: $\quad I = \bigcup_{p \in c_i} createSFCIndex(p)$;
6: $\quad totalCost = \sum_{k=1}^{|W|} minCost(q_k, I)$;
7: $\quad$ **if** $totalCost < minTotalCost$ **then**
8: $\quad\quad minTotalCost = totalCost$;
9: $\quad\quad R = I$;
10: $\quad$ **end if**
11: **end for**
12: **return** R;

If n_a is the size of the attribute set A, then the size of the power-set of A is 2^{n_a}. Since C is the set of all o-sized combinations of the power-set of A, the complexity of line 2 of the naïve algorithm is $O\left(\binom{2^{n_a}}{o}\right)$, i.e., $O\left(\frac{2^{n_a}!}{o!(2^{n_a}-o)!}\right)$. Moreover, the complexity of line 6 of the algorithm is calculated as $O\left(\binom{2^{n_a}}{o} \cdot |W| \cdot o\right)$, because the loop statement (Algo. 1: line 4) executes $\binom{2^{n_a}}{o}$ times and calculates the cost of all queries in W over a set of o indices in the worst case.

The overall worst-case complexity of the naïve index recommendation algorithm is $O\left(\binom{2^{n_a}}{o}\right) + O\left(\binom{2^{n_a}}{o} \cdot |W| \cdot o\right)$. Since the algorithm has an exponential growth in complexity, it does not scale for large attribute sets. However, it serves as a reference for the scalable index recommendation algorithms discussed in the next section.

B. Scalable Index Recommendation

In this section, we present three scalable index recommendation algorithms that deal with the complexity of the problem in two steps. The first step of each algorithm considers a limited set of attribute combinations as the initial search space. We call this set as the candidate set. The second step of each algorithm uses a certain heuristic to reduce the size of the candidate set to the user-defined limit o.

Given a workload W, the candidate set C is defined as:

$$C = \bigcup_{i=1}^{|W|} attributeCombination(q_i) - \bigcup_{i=1}^{|A|} \{a_i\}$$

Unlike the naïve recommendation algorithm, the size of the initial search space is now limited by the size of the workload. The set containing the combination of all attributes is removed from C because the final solution of each scalable index recommendation algorithm always includes an index created from the combination of all attributes in A. This index acts as a fall-back index for queries that could not be optimized.

The assumption behind the creation of the candidate set is that the workload does not contain queries using all possible combinations of the attributes in A. This assumption is realistic for typical P2P applications, where queries with certain attribute combinations are frequently issued while queries with other attribute combinations are almost never used.

Since C includes all the unique attribute combinations in W, it also represents the optimal set of indices for the queries in W. However, typically the size of C is greater than o. Any reduction in the size of C would worsen the performance of W over the set of indices created using the attribute combinations in C. Therefore, the aim of the following index recommendation algorithms is to keep the performance deterioration of W minimal, while reducing the size of C to o.

1) Cost-based Merge Algorithm: The cost-based merge algorithm recommends a set of indices by merging pairs of attribute combinations in C until the size of C is reduced to $o - 1$. Since merging any pair of elements in C increases the total cost of the workload, the idea is to merge the pairs that result in the least cost increase (c.f. Algo. 2.1 and Algo. 2.2).

Algorithm 2.1 Cost-based Merge Algorithm

1: $R = \emptyset$; $\acute{C} = C$;
2: **while** $|C| > o - 1$ **do**
3: $\quad minTotalCost = +\infty$; $totalCost = 0$;
4: $\quad x = y = -1$; $I = \emptyset$;
5: $\quad$ **for all** $c_i, c_j \in C : i \neq j$ **do**
6: $\quad\quad tempC = C$;
7: $\quad\quad tempC = tempC - \{c_i\} - \{c_j\} \cup \{c_i \cup c_j\}$;
8: $\quad\quad I = \bigcup_{c_k \in tempC} createSFCIndex(c_k)$;
9: $\quad\quad totalCost = \sum_{l=1}^{|W|} minCost(q_l, I)$;
10: $\quad\quad$ **if** $totalCost < minTotalCost$ **then**
11: $\quad\quad\quad minTotalCost = totalCost$;
12: $\quad\quad\quad x = i;\ y = j$;
13: $\quad\quad$ **end if**
14: $\quad$ **end for**
15: $\quad C = C - \{c_x\} - \{c_y\} \cup \{c_x \cup c_y\}$;
16: **end while**
17: $R = \bigcup_{c_i \in C} createSFCIndex(c_i)$;
18: $R = R \cup createSFCIndex(A)$;
19: **if** $|R| == o - 1$ **then**
20: $\quad R = addMissingIndex(R, C, \acute{C})$;
21: **end if**
22: **return** R;

The first few steps of the algorithm select a pair of elements in C, remove the selected pair from a copy of C ($tempC$), and add the union of the pair to it (Algo. 2.1: line 5-7). Next, a set of SFC-based indices is created using the modified copy of C and the cost of the workload is calculated over it (Algo. 2.1: line 8-9). These steps are repeated until a pair of elements in C that results in the least workload cost is located (Algo. 2.1: line 10-13). The pair is then removed from C and the union of the pair is added to it, making the changes to C permanent (Algo. 2.1: line 15). The merging process repeats itself as long as the size of C is greater than $o-1$. Once the size of C is less than or equal to $o-1$, the algorithm creates a set of indices R from C and adds an index created from the combination of all attributes to R (Algo. 2.1: line 17-18).

Typically, R at this point in the algorithm, represents the final set of recommended indices, but it is possible that the size of C had been reduced to $o-2$ by the previous steps of the algorithm, and therefore the size of R is $o-1$. This can happen in the case where merging a pair of elements in an o-sized C reduces C by two elements because the element produced by merging already existed in C. For example, if $C = \{\{a_1 \wedge a_2\}, \{a_2 \wedge a_3\}, \{a_1 \wedge a_2 \wedge a_3\}\}$, and the first 2 elements are merged, the size of C will be reduced by 2.

After merging pairs of elements in C, if the size of R is $o-1$ (Algo. 2.1: line 19), the algorithm invokes the procedure shown in Algo. 2.2. This procedure adds an index created using an element from the original candidate set to R.

Algorithm 2.2 $addMissingIndex(R, C, \acute{C})$

```
1: minTotalCost = +∞; totalCost = 0;
2: A = B = ∅;
3: for all c_i ∈ (Ć − C) do
4:     A = createSFCIndex(c_i);
5:     R = R ∪ A
6:     totalCost = Σ_{j=1}^{|W|} minCost(q_j, R);
7:     if totalCost < minTotalCost then
8:         minTotalCost = totalCost;
9:         B = A;
10:    end if
11:    R = R − A;
12: end for
13: R = R ∪ B;
14: return R;
```

If n_c is the size of the candidate set C, then the first loop of the cost-based merge algorithm performs $n_c - o$ executions at most (Algo. 2.1: line 2). The second loop of the algorithm (Algo. 2.1: line 5) is realized as a nested loop. Therefore, it executes $\frac{({n_c}^2-n_c)}{2}$ times, in the worst case. The third loop of the algorithm is executed while calculating the total cost of the workload over n_c indices in the worst case (Algo. 2.1: line 8). Hence, the complexity of line 8 of Algo. 2.1 is calculated as $O\left((n_c - o)(\frac{({n_c}^2-n_c)}{2})(|W| \cdot n_c)\right)$ i.e. $O\left({n_c}^4\right)$. Moreover, the complexity of line 6 of Algo. 2.2 is calculated as $O\left(n_c \cdot |W| \cdot o\right)$. Therefore, the overall worst-case complexity of the cost-based merge algorithm is $O\left({n_c}^4\right) + O\left(n_c \cdot |W| \cdot o\right)$.

2) Similarity-based Merge Algorithm: The similarity-based merge algorithm also uses merging of elements in C to recommend a set of indices. Pairs of elements that are most similar to each other, in terms of attributes, are merged until the size of C is reduced to $o-1$ (c.f. Algo. 3). Consider an example where $C = \{\{a_1 \wedge a_2\}, \{a_1 \wedge a_2 \wedge a_3\}, \{a_1 \wedge a_4\}, \{a_2 \wedge a_4\}\}$ and $o = 3$. The similarity-based merge algorithm would merge the 1^{st} and the 2^{nd} elements of C because the difference in their attributes is minimal.

Compared to the cost-based merge algorithm, the similarity-based merge algorithm is less complex because the elements in C are merged without checking them against the workload. The similarity-based merge algorithm is based on the observation that typically merging two almost identical indices will not decrease the performance of the workload significantly.

Algorithm 3 Similarity-based Merge Algorithm

```
1: R = ∅; Ć = C;
2: while |C| > o − 1 do
3:     minMergeDiff = +∞; mergeDiff = 0;
4:     x = y = −1;
5:     for all c_i, c_j ∈ C : i ≠ j do
6:         mergeDiff = |(c_i ∪ c_j) − (c_i ∩ c_j)|;
7:         if mergeDiff < minMergeDiff then
8:             minMergeDiff = mergeDiff;
9:             x = i; y = j;
10:        end if
11:    end for
12:    C = C − {c_x} − {c_y} ∪ {c_x ∪ c_y};
13: end while
14: R = ⋃_{c_i ∈ C} createSFCIndex(c_i);
15: R = R ∪ createSFCIndex(A);
16: if |R| == o − 1 then
17:    R = addMissingIndex(R, C, Ć);
18: end if
19: return R;
```

The similarity-based merge algorithm starts by selecting a pair of attribute combinations in C and calculating the difference in the attributes of the pair (Algo. 3: line 5-6). These steps of the algorithm are repeated for all pairs in C and the pair with the least difference in attributes is marked (Algo. 3: line 7-10). The marked pair is then removed from C and the union of the pair is added to it (Algo. 3: line 12). The algorithm keeps repeating as long as the size of C is greater than $o-1$. Once the size of C is less than or equal to $o-1$, the algorithm creates a set of indices R from the modified C and adds an index created from the combination of all attributes to R (Algo. 3: line 14-15). Analogous to the cost-based merge algorithm, the size of the set of indices R produced by the similarity-based merge algorithm could be $o-1$. The missing index is added to R in the similar manner, as in the cost-based merge algorithm, i.e., using Algo. 2.2 (Algo. 3: line 16-17).

If n_c is the size of the candidate set C, then the first loop of the algorithm performs $n_c - o$ executions and the second loop performs $\frac{({n_c}^2-n_c)}{2}$ executions in the worst case (Algo. 3: line 2 & 5). Therefore, the worst-case complexity of line 6 of the algorithm is $O\left({n_c}^3\right)$. Moreover, as established earlier, the complexity of Algo. 2.2 is $O\left(n_c \cdot |W| \cdot o\right)$. Therefore, the

overall worst-case complexity of the similarity-based merge algorithm is $O\left({n_c}^3\right) + O\left(n_c \cdot |W| \cdot o\right)$.

3) Selection Algorithm: The selection algorithm for index recommendation calculates the cost of the workload for each element of the candidate set C and chooses $o-1$ elements with the least cost. The idea behind the algorithm is that if the selected elements have the least cost for the workload individually, then the probability that they have the least cost for the workload altogether is also high (c.f. Algo. 4).

Algorithm 4 Selection Algorithm

1: $R = \emptyset$; $indexList[] = \{\}$; $totalCost = 0$;
2: **for all** $c_i \in C$ **do**
3: $\quad I = createSFCIndex(c_i)$;
4: $\quad totalCost = \sum_{k=1}^{|W|} minCost(q_k, I)$;
5: $\quad indexList[i] = (I, totalCost)$;
6: **end for**
7: $sortAscending(indexList)$
8: $R = \bigcup_{j=0}^{o-1} IndexList[i].getIndex()$;
9: **return** $R \cup createSFCIndex(A)$;

The selection algorithm begins by creating a SFC-based index for each attribute combination in C and calculating the cost of the workload over the created indices (Algo. 4: line 2-4). The indices along with their costs are stored in a list (Algo. 4: line 5). The list is later sorted in an ascending order of the workload cost and the top $o-1$ indices are selected into R (Algo. 4: line 7-8). Finally, an index created from the combination of all attributes is added to R, and R is returned as the set of recommended indices (Algo. 4: line 9).

The core loop of the selection algorithm performs n_c executions, where n_c is the size of the candidate set C (Algo. 4: line 2). Therefore, the complexity of the first complex statement of the algorithm (Algo. 4: line 4) is calculated as $O\left(n_c \cdot |W|\right)$. The second complex statement of the algorithm is the call to the $sortAscending()$ method (Algo. 4: line 7). A good sorting algorithm, e.g., mergesort or heapsort, has a runtime complexity of $O\left(n_c\ log\ n_c\right)$ [19]. Hence, the overall worst-case complexity of the selection algorithm is given as $O\left(n_c \cdot |W|\right) + O\left(n_c\ log\ n_c\right)$.

VI. Experimental Evaluations

We implemented the prototype of the index recommendation tool, including the index recommendation algorithms, in Java. Furthermore, we used an AMD Opteron machine with 4 GB of RAM to perform the evaluations discussed below.

Since the index recommendation tool presented in this paper is the first recommendation tool for P2P information discovery systems, it requires an evaluation using a variety of workload scenarios. Hence, a fine grained control over parameters such as total number of queries, query popularity distribution etc., is needed. Therefore, we use synthetic workloads for evaluating our tool. Moreover, unlike DBMS where benchmark workloads are made available by the TPC [20], no such workload of MAR queries is universally available for P2P systems. Workloads from P2P file sharing mostly contain multi-attribute point queries and are therefore not applicable here.

Using resource discovery in grid computing as a use-case, n number of attributes from the list shown in Table I are provided as an input to the workload generator. The workload generator creates a randomly ordered list of all attribute combinations from the provided attribute set. The list is then reduced by keeping only $r\%$ of the items and discarding the rest of them.

Attribute	Value Domain	Definition
CPU Speed	1.0 - 4.0	CPU clock speed in GHz
Busy CPU	0 - 100	Percentage of CPU(s) in use
Mem Size	1.0 - 8.0	Total Memory size in GB
Mem Used	0 - 100	Percentage of Memory in use
HDD Size	100.0 - 3000.0	Total HDD size in GB

TABLE I
Attribute List

Next, each combination in the reduced list is assigned a popularity p using the Zipfian distribution with the parameter α. α is a decimal value between 0 and 1, where 0 represents uniform distribution (all combinations have the same popularity) and 1 represents highly skewed distribution (20% combinations make up 80% of all queries). The popularity of a combination indicates the number of times a combination is repeated in the workload queries. The sum of all popularities is equal to m, where m is the total number of queries in the workload.

A MAR query is created by selecting an attribute combination from the list and randomly assigning a value range to each attribute of the selected combination. The assigned range for each attribute is chosen from the domain of the attribute shown in Table I. Finally, a workload of queries is created by selecting each attribute combination p times for value range assignment, where p is the popularity of the attribute combination.

We use an attribute set of only 5 attributes (c.f. Table I) to enable the comparison of the naïve recommendation algorithm with other algorithms, because the execution time of the naïve algorithm becomes prohibitively high for a larger attribute set.

A. Performance Evaluation

In this section, we present results from the performance evaluation of all index recommendation algorithms. For the sake of comparison, the performance of a system where an index recommendation tool is not used, i.e., a system with only a single index created from the combination of all attributes, is also evaluated. For each of the evaluation scenarios discussed below, the following two performance metrics are measured:

Total Workload Cost – Cost of all queries in the workload. The cost here refers to the estimated query cost discussed in Sec. IV, i.e., we do not consider the real cost of the workload in a DHT, e.g. the actual number of queried peers. An evaluation with a simulated DHT network follows later.

Execution Time – Execution time of an algorithm in seconds.

For each point on the graphs displayed in this section, the corresponding experiment is repeated 10 times with different workloads, and an average value is plotted.

1) Influence of Varying Attribute Combinations: In this section, we study the effect of varying the number of attribute combinations in the workload. The following values are used for the evaluation parameters described above: $m = 10000$, $n = 5$, $r = (20, 30, \ldots, 60)$, $\alpha = 0.8$, and $o = 3$. The first four parameters are used by the workload generator to produce 5 different query workloads. Each new workload has a larger variety of attribute combinations then the previous one. The last parameter used by the index recommendation algorithms, is the user-defined limit for the maximum number of indices.

With respect to the total workload cost, the naïve algorithm performs best in all cases because the algorithm recommends a set of optimal indices for the workload (c.f. Fig. 3(a)). The scalable algorithm that comes closest to the optimal solution is the cost-based merge algorithm. The similarity-based merge algorithm performs slightly worse than the selection algorithm for the cases where Algo. 2.2 is mostly not executed (for, $r = 20$ & 30). In other cases, the selection algorithm shows worse performance than the similarity-based merge algorithm. Note that the total workload cost of all index recommendation algorithms is lower than the case where only a single index with the combination of all attributes is used.

Although the execution time of the naïve algorithm is highest compared to the execution times of the other algorithms (c.f. Fig. 3(b)), it remains almost constant because the search space of the naïve algorithm always includes all possible attribute combinations (c.f. Sec. V-A). Since the initial search space of the scalable algorithms only includes the attribute combinations from the workload (c.f. Sec. V-B), the execution time of the scalable algorithms grows with increasing number of unique attribute combinations in the workload.

Fig. 3(b) also shows the limitations of the cost-based merge algorithm. If the number of attribute combinations is higher than 60%, the execution time of the cost-based merge algorithm exceeds the execution time of the naïve algorithm.

2) Influence of Varying Number of Indices: In this section, we demonstrate the effect of a varying user-defined limit for the maximum number of indices on each index recommendation algorithm. The parameter values for the evaluation are: $m = 10000$, $n = 5$, $r = 50$, $\alpha = 0.8$, and $o = (2, 3, \ldots, 6)$.

Generally, the total workload cost for each index recommendation algorithm decreases as the user-defined limit for the maximum number of indices increases (c.f. Fig. 3(c)). This happens because with increasing number of indices more queries are able to find less expensive indices for resolution.

The similarity-based merge algorithm produces higher workload cost than the selection algorithm for cases where Algo. 2.2 is mostly not executed (Fig. 3(c), for $o = 4, 5$ & 6). This indicates that the quality of indices produced by the similarity-based merge algorithm is better in cases where Algo. 2.2 is executed, because Algo. 2.2 selects the final index from the candidate set based on the total workload cost.

The total execution time of the naïve algorithm increases with increasing number of indices (c.f. Fig. 3(d)), because the complexity of the algorithm grows with increasing number of indices (c.f. Sec. V-A). However, the total execution time of the cost-based merge algorithm and the selection algorithm remains almost constant, because for these algorithms, the execution time is mostly dependent on the size of the candidate set which remains constant throughout the evaluation.

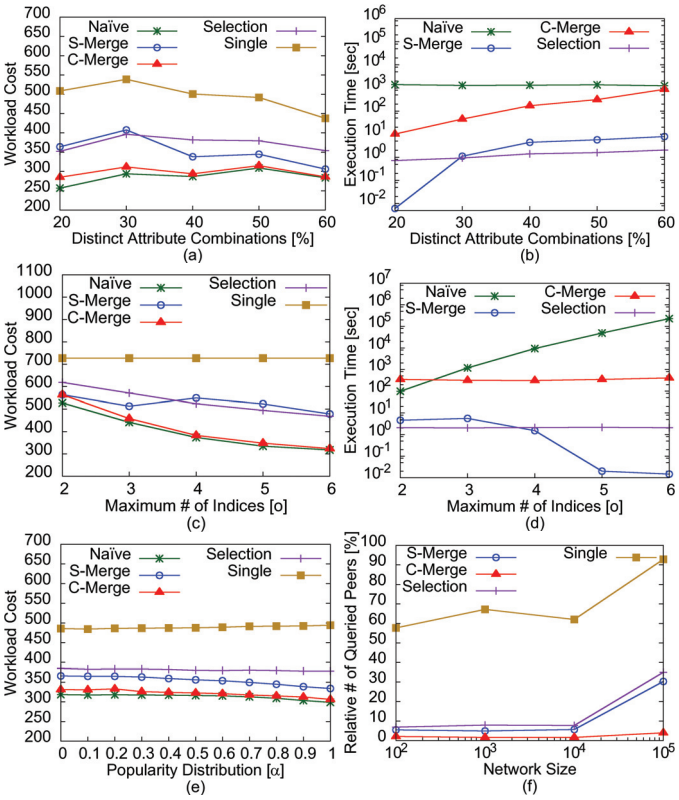

Fig. 3. Experimental Evaluations

The total execution time of the similarity-based merge algorithm suddenly decreases between values 3 and 5 for the maximum number of indices (c.f. Fig. 3(d)). This happens because in these case Algo. 2.2 is mostly not executed.

3) Influence of Varying Popularity Distribution: In this section, we illustrate the effect of varying query popularity distribution on each index recommendation algorithm. The parameter values for the evaluation are: $m = 10000$, $n = 5$, $r = 50$, $\alpha = (0.0, 0.1, \ldots, 1.0)$, and $o = 3$.

We vary the query popularity distribution of the workload from uniform distribution ($\alpha = 0$) to highly skewed distribution ($\alpha = 1$). Fig. 3(e) shows the performance of each index recommendation algorithm with respect to the total workload cost. The execution time of the algorithms is not shown because it remains almost constant throughout the evaluation, showing the same order as in the previous evaluation.

As expected, the naïve algorithm yields the least workload cost in all cases. The scalable algorithm that comes closest to the naïve approach is the cost-based merge algorithms. The selection algorithm produces the highest workload cost which slightly decreases as the popularity distribution varies from uniform to skewed distribution. As with the previous evaluation, the total workload cost of all index recommendation algorithms is lower than the case where only a single index

with the combination of all attributes is used.

Since all index recommendation algorithms generally try to include the most popular indices in the final set of recommendations, the total workload cost produced by each index recommendation algorithm decreases as the query popularity distribution varies from uniform to skewed distribution.

B. Network Simulation

In this section, we show that the SFC-based cost estimation formula presented in Sec. IV is accurate enough for a qualitative comparison between different index recommendation algorithms discussed in Sec. V. In order to do so, we first evaluate each index recommendation algorithm using the following parameter values: $m = 50$, $n = 4$, $r = 50$, $\alpha = 0.6$, and $o = 3$. Each evaluation experiment is repeated 5 times with a different workload. The estimated total workload cost (c.f. Sec. IV), averaged over 5 runs, calculated by each index recommendation algorithm, is shown in the following tables:

Algorithm	Total Workload Cost
Naïve	1.739
S-Merge	2.038
C-Merge	1.776

Algorithm	Total Workload Cost
Selection	2.084
Single	2.852

The OID System [11] is then set up in a P2P network simulation environment. Multiple SFC-based indices, corresponding to the recommendation given by an index recommendation algorithm, are defined in the OID Index Space (c.f. Sec. III). Each query from the same workload used above is then issued from a random peer in the network. This experiment is repeated 5 times for each index recommendation algorithm using the 5 corresponding workloads utilized during the evaluation above. The following metric is then measured and averaged over 5 runs for each recommendation algorithm:

Number of Queried Peers – Total number of peers queried in order to resolve all the queries in the workload.

Fig. 3(f) shows the number of queried peers for each scalable algorithm (in percentage) relative to the number of queried peers for the naïve algorithm. The relative number of queried peers are also shown for a single index created using the combination of all attributes. Fig. 3(f) asserts the same order of the algorithms as in the tables above, but now with respect to the actual number of queried peers. This shows that the cost estimation formula presented in Sec. IV is accurate enough for a qualitative comparison between different index recommendation algorithms. Moreover, Fig. 3(f) also shows that in the best case, the cost-based merge algorithm queries only 1.5% more peers compared to the naïve algorithm.

VII. Conclusion and Future Work

In this paper we presented an index recommendation tool for DHT-based information discovery systems. Given a limit for the maximum number of indices and a workload of queries, our tool recommends a set of indices that produces close-to-optimal performance for the workload queries within the given limit. The index recommendation tool consists of three scalable index recommendation algorithms: cost-based merge, similarity-based merge and selection algorithm.

Our evaluations show that there is a trade-off between the performance and the execution time of the scalable index recommendation algorithms. With respect to the performance, the cost-based merge algorithm is the best (only 1.5% worse than the naïve algorithm), generally followed by the similarity-based merge and the selection algorithms. With respect to the execution time of the algorithms, the order is reversed.

Our future work focuses on automating the process of index recommendation and installation in DHT-based information discovery systems. For that, we need to consider the overhead of installing an index compared to the benefit of installing it. Moreover, we will develop a distributed query monitoring service for gathering the workload of queries in DHTs.

References

[1] E. Meshkova, J. Riihijärvi, M. Petrova, and P. Mähönen, "A Survey on Resource Discovery Mechanisms, Peer-to-Peer and Service Discovery Frameworks," *Comput. Netw.*, 2008.

[2] J. Noh and S. Deshpande, "Pseudo-DHT: Distributed Search Algorithm for P2P Video Streaming," *Intl. Symposium on Multimedia*, 2008.

[3] F. Memon, D. Tiebler, F. Dürr, K. Rothermel, M. Tomsu, and P. Domschitz, "Scalable Spatial Information Discovery over DHTs," in *Intl. Conf. on Communication System Software and Middleware*, 2009.

[4] A. Andrzejak and Z. Xu, "Scalable, Efficient Range Queries for Grid Information Services," in *Intl. Conf. on P2P Computing*, 2002.

[5] M. Cai, M. Frank, J. Chen, and P. Szekely, "MAAN: A Multi-Attribute Addressable Network for Grid Information Services," in *Intl. Workshop on Grid Computing*, 2003.

[6] Y. Shu, B. C. Ooi, K.-L. Tan, and A. Zhou, "Supporting Multi-dimensional Range Queries in Peer-to-Peer Systems," in *Intl. Conf. on P2P Computing*, 2005.

[7] P. Triantafillou and T. Pitoura, "Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks," in *Intl. Workshop on Databases, Information Systems and P2P Computing*, 2003.

[8] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, and J. Hellerstein, "A Case Study in Building Layered DHT Applications," in *Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2005.

[9] P. Ganesan, B. Yang, and H. Garcia-Molina, "One Torus to Rule Them All: Multi-dimensional Queries in P2P Systems," in *Intl. Workshop on the Web and Databases*, 2004.

[10] C. Schmidt and M. Parashar, "Flexible Information Discovery in Decentralized Distributed Systems," in *Intl. Symposium on High Performance Distributed Computing*, 2003.

[11] F. Memon, D. Tiebler, F. Dürr, K. Rothermel, M. Tomsu, and P. Domschitz, "OID: Optimized Information Discovery using Space Filling Curves in P2P Overlay Networks," in *Intl. Conf. on Parallel and Distributed Systems*, 2008.

[12] S. Chaudhuri, M. Datar, and V. Narasayya, "Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution," *IEEE Trans. on Knowl. and Data Eng.*, 2004.

[13] N. Bruno and S. Chaudhuri, "Automatic Physical Database Tuning: A Relaxation-based Approach," in *Conf. on Management of Data*, 2005.

[14] S. Chaudhuri and V. R. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," in *Intl. Conf. on Very Large Data Bases*, 1997.

[15] ——, "Index Merging," in *Intl. Conf. on Data Engineering*, 1999.

[16] G. Valentin, M. Zuliani, D. C. Zilio, G. Lohman, and A. Skelley, "DB2 Advisor: An Optimizer Smart Enough to Recommend its own Indexes," *Intl. Conf. on Data Engineering*, 2000.

[17] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "DB2 Design Advisor: Integrated Automatic Physical Database Design," in *Conf. on Very Large Data Bases*, 2004.

[18] D. Hilbert, "Über die stetige Abbildung einer Linie auf ein Flächenstück," in *Mathematische Annalen*, 1891.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2003.

[20] *Transaction Processing Performance Council*, http://www.tpc.org/.