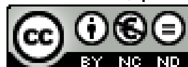
ARAN - Access to Research at NUI Galway

Provided by the author(s) and NUI Galway in accordance with publisher policies. Please cite the published version when available.

Title	Cost-Aware Processing of Similarity Queries in Structured Overlays
Author(s)	Karnstedt, Marcel; Hauswirth, Manfred
Publication Date	2006
Publication Information	Marcel Karnstedt, Kai-Uwe Sattler, Manfred Hauswirth, Roman Schmidt "Cost-Aware Processing of Similarity Queries in Structured Overlays", Proceedings of the 6th International Conference on Peer-to-Peer Computing, 2006.
Item record	http://hdl.handle.net/10379/565

Downloaded 2020-10-17T05:10:24Z

Some rights reserved. For more information, please see the item record link above.



Cost-Aware Processing of Similarity Queries in Structured Overlays*

Marcel Karnstedt, Kai-Uwe Sattler
Technische Universität Ilmenau
Germany

Manfred Hauswirth, Roman Schmidt
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Switzerland

Abstract

Large-scale distributed data management with P2P systems requires the existence of similarity operators for queries as we cannot assume that all users will agree on exactly the same schema and value representations and data quality problems due to spelling errors and typos. In this paper, we present an approach for efficient processing of similarity selections and joins in a structured overlay. We show that there are several possible strategies exploiting DHT features to a different extent (i.e., key organization, routing, multicasting) and thus the choice of the best operator implementation in a given situation (selectivity, data distribution, load) should be based on cost information allowing the system to estimate the computation and communication costs of query execution plans. Hence, we present a cost model for similarity operations on structured data in a DHT and demonstrate the efficiency of our proposal by experimental results from a large-scale PlanetLab deployment.

1 Motivation

P2P systems are starting to be recognized as tools for large-scale, decentralized information management tasks. For example, emerging applications for the Semantic and the Social Web such as metadata and index data for (specialized) search engines, catalogs, reputation data, name and directory services, etc., require *public data management* (PDM) as an enabling technology. In public data management, information, its structure, and its semantics as well as discovery and integration of data are controlled by the participants without any central control.

*The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322 and was (partly) carried out in the framework of the EPFL Center for Global Computing and supported by the Swiss National Funding Agency OFES as part of the European project NEPOMUK No FP6-027705.

A promising infrastructure approach for PDM are structured P2P overlays based on distributed hashtables (DHT) due to their good scalability and efficiency and their predictable behavior and guarantees. However, the original DHT proposals support only exact lookups for key-value pairs and are therefore too restrictive for general-purpose data management and only a few systems support more complex queries, for example, range queries in P-Grid or joins in CAN. Similarity as a concept for dealing with the expected heterogeneities both at the data and at the schema levels is not available in DHTs so far. Similarity queries are a key requirement in PDM systems for two simple reasons: (1) Data quality may be suboptimal due to spelling errors and typos and (2) we cannot assume that all users will agree on exactly the same schema and value representations, i.e., naturally people will use different though often syntactically similar conceptualizations for the same data. Therefore, we argue that similarity-based query operations play a key role in dealing with heterogeneities by enabling to retrieve data (similarity lookup and filtering) as well as to combine data (similarity join and grouping) based on fuzzy matching conditions.

Our approach for cost-aware processing of similarity queries in structured overlays, which we will present in the following, is based on a "vertical" triple storage model similar to RDF's model. This means that each tuple $(oid, v_1, \dots, v_n)$ of a given relation schema $R(A_1, \dots, A_n)$ is stored in the form of n triples $(oid, A_1, v_1), \dots, (oid, A_n, v_n)$, where oid is a unique key, e.g., a URI, and the attribute names A_i may contain a namespace prefix ns which allows the user to distinguish different relations. In order to support queries exploiting the efficient key lookup of DHTs, each triple (oid, A_i, v_i) is inserted three times into the DHT using the oid, $A_i\#v_i$ (the concatenation of A_i and v_i), and v_i as keys. This enables search based on the unique key, queries of the form $A_i \geq v_i$, and using v_i as the key for queries on an arbitrary attribute. Though this data organization produces some overhead, it has the advantages that the data are self-descriptive, i.e., do not have not to obey a predefined relation schema, and that in several DHTs, e.g., CAN or P-Grid, similar values are

stored at the same peer or neighboring peers, thus decreasing the efforts incurred in processing range queries, joins, or similarity operations.

Based on this model which was introduced in [9], we present and discuss strategies for efficient processing of similarity selections and joins in a structured overlay. We will show that there are several possible strategies exploiting DHT features to a different extent (i.e., key organization, routing, multicasting) and thus the choice of the best operator implementation in a given situation (selectivity, data distribution, load) should be based on cost information allowing the system to estimate the computation and communication costs of query execution plans. Obviously, this cannot be done in the same way as in classical centralized database systems where all necessary statistical cost information are available. Hence, we present a cost model for similarity operations on structured data in a DHT-based overlay network.

Basically our approach is generally applicable to any P2P system, be it structured, e.g., Chord, CAN, P-Grid, or unstructured, e.g., Gnutella. However, structured P2P systems have a couple of advantages, we can exploit in processing similarity queries: (1) Cost-aware processing requires the definition of accurate cost measures which in turn requires the knowledge about the complexity involved in the processing tasks, which cannot be provided for unstructured systems like Gnutella, but are available for structured systems. (2) Structured overlays, specifically DHTs, offer very low overheads for locating data items, typically $O(log(n))$. As we insert and query large amounts of small data items in our approach, this is an important factor for minimizing costs. (3) DHTs offer better data-processing related guarantees, for example, for completeness, existence, etc., which are important properties for database-like processing of predicates.

Our contribution is twofold: First, we present strategies and implementations for similarity operations as part of a distributed query engine for our query language VQL (a derivative of SPARQL), complementing and extending previous work [9]. And second, we describe a cost model enabling the system to find the most efficient query execution plan based on different operator implementations. We show the efficiency and correctness of our model by presenting results from an experimental evaluation on PlanetLab.

This paper is structured as follows: Section 2 exemplifies the type of queries we target, Section 3 defines the similarity measures we support, and Section 4 then presents the implementations of the similarity operators. Based on this, Section 5 discusses our approach to cost-based query planning. To demonstrate its efficiency, we present the results of a large-scale PlanetLab deployment in Section 6 and compare our approach with related work in Section 7 before concluding the paper.

2 Motivating Examples

In a PDM system, users will want to search for (1) data, (2) metadata, and (3) combinations of both by defining constraints on both the data and schema levels. Queries should encompass simple search conditions, but also advanced operations on the distributed data, such as joins or ranking, should be supported. To enable the user to express these types of queries, we use the VQL query language which is based on SPARQL [13], a query language for RDF. As query formulation and the logical algebra used for representing query plans are not in the focus of this paper, we will only informally introduce VQL and the logical algebra through some simple examples demonstrating its capabilities and the types of queries we discuss in this work.

Let us assume that each user in a P2P system has a movie database similar to IMDB (http://www.imdb.com/). For simplicity, without constraining generalization, we assume that the following simple relations are used by the participants:

```
movies: (title, year, type)
top100: (movie, director)
actors: (name, mtitle, rolename)
```

The basic construct of a query in VQL is a `SELECT - WHERE` block similar to SQL, but as we do not manage relations in a horizontal manner we do not have to provide a `FROM` clause. The `WHERE` clause is defined on triples (oid, A, v), selection is done using optional `FILTER(expr)` statements in the `WHERE` clause, and the functions *dist* and *edist* allow the user to express similarity in terms of distance (Euclidean or edit distance). Each term in a query starting with a question mark represents a variable and all expressions in the `WHERE` clause are implicitly combined conjunctively. Additional clauses such as `ORDER BY`, `LIMIT` and `OFFSET` are optional and have the same meaning as in SQL.

The following VQL statement defines a query for all directors who worked with actors named similar to "Billy Bill" in the years 2000-2004, including also the movie title and year in the result set, and ordered by the year the movie was produced. Additionally, we take the complicating assumption that the data provided by the users is erroneous and thus also the join operation on the movie titles, required to produce the final result uses similarity-based string matching.

```
SELECT ?d,?t,?y
WHERE { (?o1,name,?n)
        FILTER (edist(?n,Billy Bill)<3)
        (?o1,mtitle,?t) (?o2,movie,?s)
        FILTER (edist(?t,?s)<2)
        (?o3,title,?u) FILTER (edist(?t,?u)<2)
        (?o3,year,?y) FILTER (dist(?y,2002)<3)
        (?o2,director,?d) }
ORDER BY ?y DESC
```

A powerful advantage of the vertical storage model we use is the possibility to express similarity on the schema level in addition to similarity on the data level, which simplifies homogenization tasks. The following example joins data from `movies` with corresponding data from `actors`, by applying similarity first on the schema level (only `edist(title,mtitle)=1` can satisfy the filter condition on schema level in line 3) and then on instance (data) level (the actual movie titles). Moreover, to keep the final result size small, we only select those 10 movies which where produced closest to 2005 (*top-N* query).

```
SELECT ?v1,?v2,?n,?r,?y
WHERE { (?o1,?a1,?v1) (?o2,?a2,?v2)
        FILTER (edist(?a1,?a2)<2)
        FILTER (edist(?v1,?v2)<3)
        (?o1,rolename,?r) (?o1,name,?n)
        (?o2,year,?y) }
ORDER BY dist(?y,2005) LIMIT 10
```

The logical algebra used to represent the resulting operator plans is closely related to the relational algebra, but extended by some special operators.

3 Similarity Measures and Processing

The typical distance measure for numerical values is the Euclidean distance which can be mapped on range queries in the overlay network. As range queries have received quite some attention recently and several DHTs can handle them already, we will not discuss similarity on numerical attributes in further detail. For similarity measures on string values the situation is different as they cannot be mapped to range queries. This is especially true for the popular *Levenshtein distance* or edit distance [10]. Without constraining the general applicability of our approach, we will focus on the processing of single string similarity predicates based on the edit distance $edist(s_1, s_2)$ (our approach works for any distance measure $d(x, y) \rightarrow \Re$).

In its simplest form the edit distance of two strings s_1 and s_2 is the number of operations (insertion, deletion or substitution of characters) needed to transform s_1 into s_2. For instance, the edit distance of "edna" and "eden" is 2. Several approaches exist to efficiently process similarity measures based on the edit distance. We base our work on that of Navarro et al. [11] and Gravano et al. [6] who suggest the evaluation of the edit distance using substrings of fixed length q, so-called *q-grams*. We briefly discuss the main results of these works which we exploit in our approach. The main observation is that if we pick any $d + 1$ non-overlapping q-grams, at least one of them must be fully contained in the comparison string (the two matching q-grams *correspond* to each other) [11].

As a consequence, we extend the original storage scheme highlighted in Section 1 as follows: Rather than indexing only whole strings, we additionally split them into q-grams and index those (both on the instance and on schema levels), i.e., for a triple $t = (oid, A_i, v_i)$ we store the following in the DHT:

$$[h(oid), (oid, t)], [h(A_i\#v_i), (oid, t)], [h(v_i), (oid, t)], \\ [h(A_i\#qg_1(v_i)), (oid, t)], \ldots, [h(A_i\#qg_n(v_i)), (oid, t)], \\ [h(qg_1(v_i)), (oid, t)], \ldots, [h(qg_n(v_i)), (oid, t)], \\ [h(qg_1(A_i)), (oid, t)], \ldots, [h(qg_m(A_i)), (oid, t)]$$

($h()$ denotes the hash function of the DHT to generate the key under which the triple is stored, and $qg_i(s)$ denotes the ith q-gram of s). This storage scheme involves a non-negligible overhead (depending on the actual choice of indexed attributes), but decreases query processing costs considerably, as we will show in the following sections.

As an example, consider a tuple `t: {123,edna}` with schema `s: {id,name}`. In the original scheme the following data items would be stored in the DHT (we assume that the triple's OID is 1):

[h(1), (1,id,123)], [h(id#123), (1,id,123)],
[h(123), (1,id,123)],
[h(1), (1,name,edna)], [h(name#edna), (1,name,edna)],
[h(edna), (1,name,edna)]

Extending this by a 3-gram index on instance level of attribute `name` produces the following additional data items to be stored:

[h(name#edn), (1,name,edna)],
[h(name#dna), (1,name,edna)],
[h(edn), (1,name,edna)], [h(dna), (1,name,edna)]

Additionally, indexing on schema level results in the following additional data items to be stored:

[h(nam), (1,name,edna)], [h(ame), (1,name,edna)]

4 Physical Operators

Having discussed the basic conceptual approach for similarity queries, we will now describe the implementations of similarity operators available in our query engine. As already mentioned, we distinguish between queries on instance level, on schema level, and queries combining both levels. Due to space constraints we will only describe the processing of queries on instance level as the handling of queries on schema level differs only in the part of the triples which is processed. We start with similarity selections as the basis for advanced operators and then present similarity joins and ranking operators as examples for advanced operators. In the following we will only deal with string similarity as numerical similarity measures can simply be mapped on range queries as argued in Section 3 and additionally, string data will be the dominating data type in most systems (not only on schema, but also on instance level).

In principle, we could process string similarity queries by only utilizing the functionality already provided by a DHT. By issuing key lookups, we can locate the data concerned by similarity predicates, e.g., by prefix searches on the attribute names. However, this would be very expensive as instance level queries can result in involving the whole overlay if popular attributes are distributed among all peers, e.g., as in Chord. If this is combined with similarity measures on the schema level the situation would be even worse as we have to look at even more data using this simplistic approach. Additionally, only a fraction of the queried peers will actually contribute to the final result. We will denote this simple strategy as *term-based processing* which will serve as a basic means of comparison to show the gain we can achieve through our *q-gram-based processing* strategy. This strategy exploits additional indexes based on q-grams as described above and incurs additional messages for querying these indexes, but saves a lot of bandwidth and message costs for processing the queries in most cases.

To be able to compare the costs of these two alternatives we will define a cost model in Section 5. However, it is not the focus of this paper to redefine optimization and planning tasks already known from relational and distributed database systems. Rather we target the costs incurred by the actual gathering of data distributed among the peers of the overlay network which is required to be able to process similarity queries.

4.1 Similarity Selection

The most fundamental operation we have to support in processing similarity queries is *similarity selection*, which means that all data corresponding to a similarity predicate is located and returned to the peer having initiated the query. With term-based processing, we contact *each* peer responsible for a part of the data to be checked as shown in Algorithm 1 for basic similarity predicates such as $edist(A, s) < d$, where A is a given attribute name, $h()$ is the hashing function used by the overlay, p is the peer to send the query to (determined by the routing algorithm of the used overlay), s is the search string, and d is a positive integer denoting the edit distance.

Algorithm 1 Term-based similarity selection $TSel(s, A, d, p)$

1: $T = SimRetrieve(h(A), p, s, d)$;
2: $R = \emptyset$;
3: **for all** $t \in T$ **do**
4: $\quad R = R \cup \bowtie_{OID}; \{Retrieve(h(\xi(t, 1)), p))\}$
5: **end for**

Assuming that $Retrieve(key, peer)$ is the normal query forwarding and search function of an overlay, $SimRetrieve$ extends it with similarity search functionality, i.e., the normal routing is not touched but each peer receives the search string s to be used for similarity matching plus the required similarity d for local evaluation. We also assume that $Retrieve$ can do both exact and prefix (path) queries, for example, as in P-Grid [1]. Thus a query for $h(A)$ would be successful although we actually indexed $A\#v$ as described in Section 1. This is just a shortcut to exploit existing overlay functionality and to be more efficient in query processing. For systems not offering prefix search, triples (oid, A_i, v_i) would be indexed with A_i as the key. However, in the following we implicitly assume that prefix search is supported by $Retrieve$ and thus also by $SimRetrieve$, without constraining the general applicability of the algorithms. As we are dealing not only with triples, but also with tuples, $\bowtie_{OID}$ uses the OID to retrieve all parts of a tuple and reconstruct it (this is equivalent to $Retrieve(key(\xi(t, 1)), p))$, where $\xi(t, i)$ simply means to take the ith field of a tuple t, which in our storage model is the OID for $i = 1$). The result of this operation is then collected in R. If we would only work with triples, e.g., RDF, this step would not be necessary.

The q-gram based variant of similarity selection is shown in Algorithm 2.

Algorithm 2 Q-gram-based similarity selection: $QSel(s, A, d, p)$

1: determine $d + 1$ q-grams Q from s;
2: $R = \emptyset$;
3: **for all** $q \in Q$ **do**
4: $\quad T = SimRetrieve(h(A\#q), p, s, d)$;
5: $\quad$ **for all** $t \in T$ **do**
6: $\quad\quad R = R \cup \bowtie_{OID} \{Retrieve(h(\xi(t, 1)), p))\}$;
7: $\quad$ **end for**
8: **end for**

The main difference to the term-based variant is that $SimRetrieve$ is called in a loop, once for each q-gram. Then, we again reconstruct the tuples (line 6) by querying for all found OIDs. Algorithm 2 is actually a revised version of the algorithm suggested in [9]. The same paper also proposes some improvements differing in the application of concrete filtering steps and the amount of state which needs to maintained in the query processing. This improved variant only differs slightly from Algorithm 2:

DelSel: Instead of gathering temporary results in line 4, peers responsible for q-grams delegate the query to peers responsible for corresponding OIDs, and those reply to the initiating peer.

The impact of this variant depends on the current network state and data distribution, and is covered by our cost model which we present in Section 5.

4.2 Similarity Join

A similarity join is one of the most important similarity operators as it is a powerful tool to overcome heterogeneity

at the schema level, which allows the system to deal with semantic inconsistencies, i.e., supports schema integration, and at the data level to address inconsistencies or inaccuracies in the data to be processed. The following discussions are based on the definition of similarity string joins given in [5]: Given two input sets of tuples r and s with schemas $\hat{r} : (X_1 \dots X_k)$ and $\hat{s} : (Y_1 \dots Y_l)$ a similarity join produces the cross product of all tuples and returns those tuples t with schema $\hat{t} := (X_1 \dots X_k Y_1 \dots Y_l)$ for which a similarity predicate $p : edist(X_i, Y_j) < c, i \leq k \wedge j \leq l \wedge c$ is constant, is true.

Such joins are also conceivable only on schema level, but we expect joins comprising both levels to occur much more frequently. This corresponds to similarity predicates like $edist(\hat{A}, \hat{B}) < c_1 \wedge edist(A, B) < c_2$ (where $\hat{A}$ denotes the attribute's name, rather than its content A). In the following we discuss similarity joins on the instance level as the same algorithms can be applied on the schema level and a combination of the two levels then is straight-forward. Figure 1 shows part of an operator plan we have to handle for a similarity join on instance level.

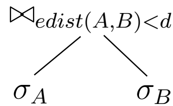


Figure 1. Join part of an operator plan

To process such a join, three basic approaches exist:

1. Process σ_A and σ_B separately and evaluate the join on the data gathered locally. The disadvantage of this strategy is that a lot of data may be transferred unnecessarily which will not contribute to the result.
2. Process σ_A, i.e., *materialize* data for σ_A (w.l.o.g. we expect the left side to be materialized) and apply a nested loop approach for querying similar data from the right side.
3. Include both selections into the join processing: a peer responsible for object(s) from the left side delegates the query to peer(s) responsible for the right side, similar to the standard approach of *mutant query plans* [12].

Term-based processing implies to gather all data needed at the query initiator and process everything locally. This corresponds to the first variant above and involves the execution of two (similarity) selections, which again can be varied in the actual way of processing.

Assuming that the left input set is materialized completely, we can use the approach of processing string similarity based on q-grams in order to find matching candidate tuples from the right side, before completely fetching all of the corresponding strings (variant 2 above). An intuitive implementation is based on a (block) nested loop access, which means that each materialized tuple from the left side is used as input for a corresponding similarity selection on the right side (not single tuples, but actually block(s) of them, respectively). Algorithm 3 illustrates this approach.

Algorithm 3 Nested Loop Similarity join: $NLJoin(A, B, d, p)$

1: $L = Retrieve(h(A), p)$;
2: $R = \emptyset$;
3: **for all** $t \in L$ **do**
4: $\quad R = R \cup \{\bowtie(t, t') : t' \in QSel(\xi(t, A), B, d, p))\}$;
5: **end for**

To get a more complete view, we also included the left-side selection (here exemplarily by calling $Retrieve$ in line 1), though this is a separate operator. The actual join tuples are built in line 4: Each tuple from the left side is joined with all similar tuples from the right side (located by calling $QSel$) by issuing $\bowtie$. The final result is collected in R. The somehow "centralized" character of this method allows for minimizing the repeated querying of duplicate strings and q-grams. This can be achieved by merging the single queries into multiple blocks (or only one single block). This eliminates the disadvantage of variant 1, but still puts the join processing load on a single node.

An interesting alternative for distributed environments as P2P systems is to include both materializing operations into the join operator, rather than only the left one (variant 3 above). In this case, peers responsible for parts of the left side delegate directly to right side peers: $DelJoin(Retrieve(key(A), p), p, B, d)$. $DelJoin$ calls $Retrieve$ as introduced before, but responsible peers do not return results directly. Rather, they forward similarity selections to the peers responsible for the right side. These peers reply to the peer referenced by the second parameter p (the first p only applies to $Retrieve$), if any tuples are actually joined according to $edist(A, B) < d$. This variant may be extended to a block-based processing at each involved peer, similar to Algorithm 3. The advantage of this method is, that we do not include any waiting states in the processing. The disadvantage is that less opportunities exist to eliminate repeated querying of identical strings and q-grams, as outlined for Algorithm 3.

4.3 Ranking Operators

In large-scale environments, like, for example, Google, where only best-effort solutions are applicable, ranking queries are a necessary query type. As an example, we discuss top-N similarity queries. Top-N queries are always based on a certain ranking function. We describe the implementation of a nearest neighbor ranking (NN), though other rankings are supported, but rather unsuitable for string processing.

For the term-based variant we simply query for the attributes needed and determine the top-N strings locally at

the initiating peer. The q-gram-based version relies on the predetermination of an interval to query. This interval is determined such, that it potentially comprises all N needed tuples. If not, this interval is successively extended until at least N objects are available locally. Algorithm 4 illustrates the method.

Algorithm 4 Top-N Query: $TopN(s, A, N, p)$

1: $c = |\{d \in \delta(p) : h(d) \supseteq h(A)\}|$;
2: {determine the size r of the local range of A;}
3: $range = N/\frac{c}{r} = N \cdot \frac{r}{c}$;
4: $d = DetIntv(range, s, 0)$;
5: $R = \emptyset$;
6: **repeat**
7: $R = R \cup QSel(s, A, d, p)$;
8: $range = N/\frac{|R|}{2 \cdot d} = N \cdot \frac{2 \cdot d}{|R|}$;
9: $d = DetIntv(range, s, d)$;
10: **until** $|R| \geq N$
11: $R = Limit(Sort(R, A), N)$;

First each peer determines the number of local data elements for attribute A. Based on this number the peer calculates a data density which approximates the number of values from A that are stored at a single peer. If the data is distributed among peers load balanced, this density is a good choice for determining an according interval to query first. If we have to adapt the interval we calculate a new data density based on values actually retrieved.

5 Cost-Based Planning

The crucial part about optimizing query plans is how to obtain the data in the distributed case, i.e., how to access the corresponding peers in the DHT in an optimal way without unnecessary (re-)transmissions. If data is available locally, cost estimation for operators is identical to the relational case. The main possible alternatives for obtaining data are (1) to collect all data and process operators locally and (2) to use the indexes defined in the underlying DHT as access structures in a way that minimizes data transmission and query costs. In this section we discuss the problem of cost estimation for the introduced physical operators in order to be able to compare the different variants and enable a DHT to choose the optimal query plan.

In a distributed environment the main cost measures are the number of messages m and the number of hops h needed to process queries. Low bandwidth consumption and short query answer times may also be of interest, but are very hard (if not impossible) to predict. Luckily, the number of query hops usually reflects query answer times. In the following we provide formulas to estimate m and h for each of the introduced operators. This builds on estimating costs for processing a single lookup query in the overlay system. Usually, a limit logarithmic in the number of peers N for m and h is guaranteed. In the following we will refer to m_l for the number of messages a lookup results in, and h_l for the number of hops, respectively. We will only consider query messages, no system messages or answers transmitted between peers directly.

We provide knowledge about string and q-gram selectivities by managing local indexes on each peer for approximating data distribution. One possibility is using tries as in [14]. In the following we expect all needed values to be available at each peer. In a real-world environment some of these values will have to be approximated.

operator op	m_{op}	h_{op}
$Fetch(A)$	$m_l + r_A - 1$	$h_l + r_A - 1$
$TSel(s, A, d)$	$m_{Fetch}(A)$	$h_{Fetch}(A)$
$QSel(s, A, d)$	$(d + 1) \cdot m_l$	h_l
$DelSel(s, A, d)$	$(d + 1) \cdot m_l$	h_l
$TJoin(A, B, d)$	$m_{LeftSel}(A) +$ $m_{Fetch}(B)$	$max(h_{LeftSel}(A),$ $h_{Fetch}(B))$
$NLJoin(A, B, d)$	$m_{LeftSel}(A)$ $+$ $c_{left} \cdot m_{QSel}(s_A, B, d)$	$h_{LeftSel}(A)$ $+$ $h_{QSel}(s_A, B, d)$
$DelJoin(A, B, d)$	$m_{LeftSel}(A)$ $+$ $c_{left} \cdot m_{QSel}(s_A, B, d)$	$h_{LeftSel}(A)$ $+$ $h_{QSel}(s_A, B, d)$

Table 1. Costs for physical operators

Table 1 summarizes the formulas for cost estimation. The *Fetch* operation was not introduced separately. This operator provides the functionality of the first step in $TSel$, where all values of a single attribute are fetched. This is processed in a sequential way: a first query is sent to one peer responsible for a part of that attribute. This peer returns matching data and forwards the query if there are other peers which are responsible. This processing is repeated until the last responsible peer is queried. We have to effort m_l messages to reach the first peer, and $r_A - 1$ messages forwarded to next peers, if r_A represents the number of peers responsible for attribute A and key data is clustered according to A. Instead of fetching a whole attribute, the q-gram-based similarity selection $QSel$ queries for $d + 1$ q-grams in parallel.

If c_A represents the number of unique values in A and sel_s the selectivity of the predicate $edist(A, s) \leq d$, we have to query for $c_A \cdot sel_s$ complete tuples in the final step of both selections, resulting in $c_A \cdot sel_s \cdot m_l$ messages. This is a factor equal for all implementations if the tuples are materialized finally. Thus, the formulas do not include these costs for tuple materialization. Subqueries for this operation can be processed in (quasi-)parallel, which results in h_l hops.

The main impact on performance between $TSel$ and $QSel$ lies in r_A. A term-based selection gets the more expensive the more peers are responsible for a part of the queried attribute. Furthermore, performance of $TSel$ strongly depends on the current network state, because of its sequential character. The last of the introduced similarity selection operators is particularly suited for dynamic environments, because there are no temporary answers to and no

waiting state at the initiating peer. This is at the expense of more query messages, as several tuples will be queried multiple times for materialization (issued by different peers).

The costs for advanced operators are estimated on the basis of the costs of similarity selections. $LeftSel(A)$ symbolizes any suitable (similarity) selection on attribute A. The nested loop similarity join promises to be efficient if $LeftSel(A)$ refers to a selection that reduces the size c_{left} of the left input, e.g., a similarity selection $QSel(s_A, A, d_A)$. In $TJoin$ the right input is always fetched completely. Another main difference should to be found in the consumed bandwidth, as $TJoin$ completely fetches attribute B and $NLJoin$ usually only a small fraction. Similar to $QSel$ and $DelSel$, the third join implementation $DelJoin$ differs in the number of answer messages and the economized waiting state at the query initiator.

6 Evaluation

We implemented our algorithms on top of the Java-based P-Grid DHT and performed first experiments on PlanetLab [4]. The aim of these experiments was to evaluate bandwidth consumption and number of messages as the key performance characteristics. Furthermore, we give a first proof of concept for the introduced cost model by comparing estimated costs to the real costs.

6.1 Experimental setup

In the experiments we used a network of approximately 400 peers each running on a dedicated physical PlanetLab node. Each node inserted 10 strings of lengths between 8 and 45 characters, randomly chosen from a 4000 entry sample of movie titles from the IMDB database with a skewed heavy-tail key distribution as shown in Figure 2 (log-log scale). The figure only shows keys which were inserted more than once and highlights a power-law like key distribution, as it is usual in DHT-based systems when working on string data. The figure provides a general view on the actual data distribution and thus, gives directions for selectivity estimation on strings (and q-grams, in particular), which is used in the introduced cost model. With all q-grams and replication (average replication factor: 5) each peer was responsible for approximately 900 index entries. The constructed P-Grid tree had a height of 8.

We implemented all of the introduced physical operators. In Section 5 we already discussed possible problems of several processing strategies in dynamic environments. The term-based operators can result in involving a main part of the peers in the overlay system into the processing of a single similarity selection. As a consequence we encounter a huge amount of messages and heavy load, which results in poor answer times and can bring peers to crash. Moreover,

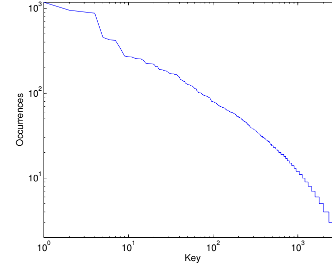


Figure 2. Key distribution (log-log scale)

the sequential character of the strategy results in poor performance in general, and processing of a query may not be finished at all, as crashed peers can interrupt a corresponding sequence of queried peers completely. We experienced these symptoms in the real-world environment of PlanetLab and could not achieve useful results with the described experimental setup. Thus, we can only provide cost estimations for these operators, but these reflect the bad performance of the approach. For the future, we plan to improve this strategy by applying parallelized routing techniques and extended usage of acknowledgment messages in order to overcome the performance problems.

To evaluate other operators we used similarity queries which affected data from all partitions of the data set. A set of 5 randomly chosen strings was queried in distance 3 using the q-gram based similarity selections. We extended the query mix by 5 similarity string joins. The left input of these joins was provided by a q-gram based selection in distance 1. We set the actual join distance for tuples from the right to 3. Each peer initiated a randomly generated query mix like this by starting a query every 5-8 minutes. All of the following figures show the average number of messages, average bandwidth consumption, respectively, measured per minute at each peer.

6.2 Experimental results

Figure 3 shows the measured number and estimated number of messages for the similarity string joins.

From time 340-380 (time 0-340 was used to bootstrap the P-Grid overlay system) we ran the $NLJoin$ operator and predicted corresponding costs. From time 380-420 the plot of estimated costs also shows the estimations for the term-based similarity join. As explained before, we did not achieve useful results with this operator in the described experimental setup. But, this is anticipated by the estimated costs as well. The performance of the nested loop join behaves as expected and also as estimated. The small peaks signalize materialize operations of tuples contained in the final result. These subqueries result in several extra messages, because object IDs are spread all over the system. As we want to determine the correct relations between costs of

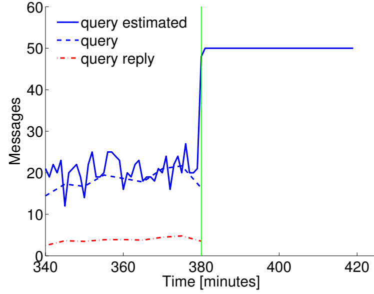


Figure 3. Real and estimated costs for $NLJoin$ (time 340-380) and $TJoin$ (time 380-420)

different physical implementations, rather than exact costs, these results are fine. The plot shows that we are able to achieve this.

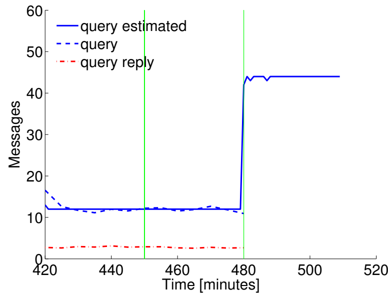


Figure 4. Real and estimated costs for $QSel$ (time 420-450), $DelSel$ (time 450-480) and $TSel$ (time 480-510)

In Figure 4 we present analog results for similarity selections. Time 420-450 corresponds to the execution of $QSel$, time 450-480 to $DelSel$, and time 480-520 to the term-based selection $TSel$ (again, only predicted costs are plotted). The estimated costs are constant, because we only involve the processing of actual queries into the calculation (no query replies). This is reflected by the real costs, small fluctuations are due to the following materialize operations. Again, the burst in the plot of estimated costs reflects the bad performance of the term-based similarity selection. As estimated, the costs of $QSel$ and $DelSel$ are almost identical. Similar to the experiments on joins, the plots show the correctness of our cost model and that the bad performance of the implemented term-based selection is predictable a-priori.

Finally, we illustrate the bandwidth consumption of both query types in Figures 5 and 6. These figures show that, despite the storage overhead we experience, the consumed bandwidth is within an acceptable range. The tested selection types in Figure 6 show little difference between bandwidth used for queries and bandwidth used for query

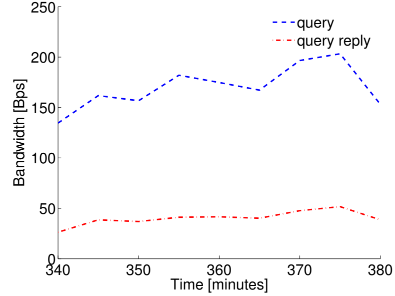


Figure 5. Bandwidth consumption for similarity join

replies. The plots also show the higher bandwidth consumption of join queries in contrast to selections. This conforms with our expectations, because the join operators combine selection operators with additional queries.

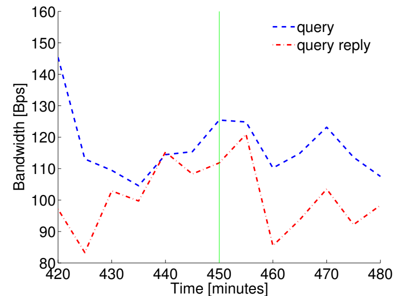


Figure 6. Bandwidth consumption for similarity selections

As a consequence, we experience the q-gram based algorithms as an importing extension to known overlay systems in order to process similarity queries efficiently. In contrast to traditional implementations this approach scales up to high numbers of peers and is applicable in dynamic environments with skewed data distributions. The costs of these operators can be estimated in an easy fashion, allowing optimizers to speed up processing and lower network load. Such estimations are based on small information, but are quite accurate, which is reflected by the correct relations of estimated costs for analog operators implemented differently.

7 Related Work

There is a number of related approaches aiming to support complex structured queries on DHT-based data management solutions. Strategies for implementing classical relational algebra operators and particularly joins on top of a CAN-based overlay have been developed as part of the PIER project [7, 8]. Query operators such as equi-selection,

range selection and hash joins, and their implementation, using modified Chord search algorithms are presented in [16]. Extensions of this work address range queries and load balancing as well as replication using a so-called multi-rotation hashing.

The approach presented in [3] exploits a similar data organization for RDF data as in our work, but does not address similarity-based queries. Several other approaches considering range queries already as similarity queries exist. However, we consider here only systems that support already a more complex notion of similarity, such as top-N queries and similarity on textual data.

Compared to all above approaches, our approach supports a larger set of query types on structured data, i.e., top-N queries with different ranking functions, similarity joins on schema and instance level as well as other advanced similarity queries, and allows us to combine these with standard relational algebra operators.

LSH forest [2] uses a locality-sensitive hashing (LSH) function to index high-dimensional data for answering (approximate) similarity queries. The queries return the m points in the data set closest to the query according to a distance function. The system is based on P-Grid and stores documents in the overlay network using the LSH function. Therefore, similarity queries can be performed by first routing to the peer closest to the initial query and then returning documents similar to the query by using existing neighbor links in P-Grid. The paper does not provide an evaluation of required messages or bandwidth as provided by us.

EZSearch [15] is based on the Zigzag hierarchy which clusters semantically close nodes in a multi-layer hierarchy and supports range queries and top-N queries. The evaluation of the system by a simulation shows that the system works well for both query types even for Zipf-like query distributions, but it remains unclear how the system deals with skewed data distributions, which require sophisticated load balancing mechanisms. Additionally, no experimental evaluation exists.

8 Conclusions

In this work, we presented similarity queries in DHTs as a key functionality for evolving structured overlays into a viable infrastructure for large-scale public data management and information retrieval. Similarity queries on schema and instance levels are a basic requirement to overcome semantic inconsistencies and poor data quality, and we specifically discussed similarity queries, joins and top-N queries as the major relevant operators. Our approach elegantly combines the efficiency of structured overlays with the flexibility of similarity query processing. The cost model included in our approach allows the system to predict the costs of the available physical operators dynamically to choose the optimal one for the current network conditions. We discussed various physical operator implementations and demonstrated the efficiency of our approach with experimental results from a large-scale PlanetLab deployment.

References

[1] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *CoopIS*, pages 179–194, 2001.

[2] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.

[3] M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW*, pages 650–657, 2004.

[4] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[5] W. W. Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Trans. Inf. Syst.*, 18(3):288–321, 2000.

[6] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[7] M. Harren, J. Hellerstein, R. Huebsch, B. Thau Loo, S. Shenker, and I. Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *IPTPS*, pages 242–259, 2002.

[8] R. Huebsch, J. M. Hellerstein, N. Lanham, B. Thau Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, pages 321–332, 2003.

[9] M. Karnstedt, K.-U. Sattler, M. Hauswirth, and R. Schmidt. Similarity Queries on Structured Data in Structured Overlays. In *Int. Workshop on Networking Meets Databases (NetDB'06) icw ICDE*, pages 32–37, 2006.

[10] V. Levenshtein. Binary codes of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[11] G. Navarro and R. A. Baeza-Yates. A Practical q -Gram Index for Text Retrieval Allowing Errors. *CLEI Electron. J.*, 1(2), 1998.

[12] V. Papadimos and D. Maier. Mutant Query Plans. *Information and Software Technology*, 44(4):197–206, April 2002.

[13] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF. W3C Candidate Recommendation 6 April 2006. http://www.w3.org/TR/rdf-sparql-query/.

[14] E. Schallehn, I. Geist, and K. Sattler. Supporting Similarity Operations based on Approximate String Matching on the Web. In *CoopIS*, pages 227–244, 2004.

[15] D. A. Tran. Hierarchical Semantic Overlay Approach to P2P Similarity Search. In *USENIX Technical Conference*, pages 355–358, 2005.

[16] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *DBISP2P'04 icw SIGMOD*, pages 169–183, 2004.