Similarity Queries on Structured Data in Structured Overlays*

Marcel Karnstedt†, Kai-Uwe Sattler†, Manfred Hauswirth‡, Roman Schmidt‡
†Department of Computer Science and Automation, TU Ilmenau, Germany
‡School for Computer and Communication Sciences, EPFL, Switzerland

Abstract

Structured P2P systems based on distributed hash tables are a popular choice for building large-scaled data management systems. Generally, they only support exact match queries, but data heterogeneities often demand for more complex query types, particularly similarity queries. In this work, we suggest a vertical data organization, which allows for efficient processing of similarity queries on instance as well as on schema level, and we introduce corresponding physical similarity operators. Our novel approach is shown to be suitable in conjunction with P-Grid, as an example of robust, large-scaled and self-organizing P2P systems.

1 Motivation

P2P systems have been gained much attention not only as a platform for file sharing or as a new paradigm in distributed systems, but also by the database community. Beside P2P-based IR systems and peer data management systems (PDMS) as an extension of federated database systems, *structured* P2P systems based on distributed hash tables (DHT) are a very promising approach for large-scale distributed data management. The basic idea of these systems is to map a key space to a set of peers such that each peer is responsible for a given region of this space and storing data whose hash keys pertain to the peer's region. The various proposals for DHTs differ mainly in the topology, e.g., CAN is based on a grid, Chord uses a ring, and P-Grid which we use in our work is based on a prefix tree.

The main advantage of structured P2P systems compared to unstructured ones is their deterministic behavior – in most approaches search complexity is guaranteed to be logarithmically – and the fair balancing of load among the peers (assuming an appropriate hash function). However, the original DHT proposals support only exact-match lookups for key-value pairs. Only a few approaches address more complex queries, e.g., joins in PIER [8], substring, range queries and path queries in P-Grid [1, 6] or our own work [10]. Supporting such queries is an important prerequisite for a novel kind of applications for DHTs: public data management. By public data we mean structured data which is collected, maintained, and used by a large community in a distributed and fair fashion. Examples of public data are name and directory services, e.g., LDAP, UDDI, metadata and index management (e.g., for the Semantic Web), search engines, or scientific data.

The main challenges in this context are *(i)* data organization both on the logical level (schema) and on the physical level (fragmentation, indexing) and *(ii)* processing complex structured queries.

In this paper, we address these challenges by using an extensible vertically-oriented data organization scheme for public data management based on the idea of RDF and a query language exploiting this scheme. Furthermore, we argue that due to the inherent heterogeneities of public data (both on instance and schema level) similarity-based operators are required whose implementation on the P-Grid DHT and the data organization scheme are presented as well.

2 The P-Grid Overlay Network

The approach presented in this paper uses the P-Grid [1, 2] distributed hash table (DHT). We assume that the reader is familiar with the general concepts of DHTs and will thus only address the specific and relevant properties of P-Grid.

In P-Grid peers refer to a common underlying binary trie structure to organize their routing tables. Data keys are computed using an order-preserving hash function to generate keys and without constraining general applicability we use binary keys in P-Grid. Each peer constructs its routing table such that it holds peers with exponentially increasing distance in the key space from its own position. This technique basically builds a small-world graph [9], which enables search in $O(\log N)$ steps. Each peer $p \in P$ is associated with a leaf of the binary trie, i.e., a key space partition, which corresponds to a binary string $\pi(p) \in \Pi$ called the peer's *path*. For search, the peer stores for each prefix $\pi(p,l)$ of $\pi(p)$ of length l a set of references $\rho(p,l)$ to peers q with property $\overline{\pi(p,l)} = \pi(q,l)$, where $\overline{\pi}$ is the binary string π with the last bit inverted. This means that at each level of the trie the peer has references to some other peers that do not pertain to the peer's subtrie at that level which enables the implementation of prefix routing.

Each peer stores a set of data items $\delta(p)$. For $d \in \delta(p)$ $key(d)$ has $\pi(p)$ as prefix but it is not excluded that tem-

*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.

porarily also other data items are stored at a peer, that is, the set $\delta(p, \pi(p))$ of data items whose key matches $\pi(p)$ can be a proper subset of $\delta(p)$. Moreover, for fault-tolerance, query load-balancing, and hot-spot handling, multiple peers are associated with the same key-space partition (structural replication), and peers additionally also maintain multiple references $\sigma(p)$ to peers with the same path (data replication). P-Grid supports exact and substring search (Algorithm 1 shows the procedure) and range queries on keys. As P-Grid's order-preserving hashing clusters related data items, range queries can be implemented very efficiently [6] which is important for the presented approach.

Algorithm 1 Search in P-Grid: $Retrieve(key, p)$

```
1: if π(p) ⊆ key or π(p) ⊃ key then
2:    return({d ∈ δ(p)|key(d) ⊇ key});
3: else
4:    determine l such that π(key, l) = π(p, l)‾;
5:    r = randomly selected element from ρ(p, l);
6:    Retrieve(key, r);
7: end if
```

p in the algorithm denotes the peer that currently processes the request, $\subseteq$ refers to the substring relation. The algorithm always terminates successfully, if the P-Grid is complete (ensured by the construction algorithm) and at least one peer in each partition is reachable (ensured through redundant routing table entries and replication). Due to the definition of ρ and $Retrieve(key, p)$ it will always find the location of a peer at which the search can continue (use of completeness). With each invocation of $Retrieve(key, p)$ the length of the common prefix of $\pi(p)$ and key increases at least by one and therefore the algorithm always terminates. As P-Grid uses a binary trie, *Retrieve(key)* is of complexity $O(\log |\Pi|)$, measured in messages required for resolving a search request, in a balanced trie. Though skewed data distributions in connection with order-preserving hashing may imbalance the trie, [1] shows that due to the randomized choice of routing references from the complimentary subtrie, the expected search cost remains logarithmic ($0.5 \log N$), independently of how the P-Grid is structured. Additionally, P-Grid's construction algorithm [2] keeps imbalances minimal anyway.

3 Storing and Querying Structured Data

As motivated in the introduction our goal is to store structured (i.e., relational) data in a DHT in a flexible way. In previous work [10] we used the "classical" approach of horizontally-oriented tuples: A peer stores the whole tuple $(oid, v_1, \ldots, v_n)$ where the peer is determined by applying the hash function $key(oid)$. Range queries are supported by a locality-sensitive scheme, i.e., tuples of a certain relation are stored along a space-filling curve. However, the approach requires that all users agree on a common database schema. An alternative approach is a vertically-oriented organization. From a conceptual point of view, this means that each tuple $(oid, v_1, \ldots, v_n)$ of a relation schema $R(A_1, \ldots, A_n)$ is stored in the form of n triples $(oid, A_1, v_1), \ldots, (oid, A_n, v_n)$.

For our purpose, we assume that oid is an unique value, e.g., a URI, and the attribute names A_i may contain a name space prefix ns which allows us to distinguish between different relations. Furthermore, null values are not explicitly represented. Obviously, this scheme follows the idea of RDF triples but has also been proposed in other contexts. Storing these triples in a DHT raises the question which element should be used for hashing. Here, *(a)* object lookups are supported if the hash function is applied to oid, *(b)* selections of the form $A_i \geq v$ can be performed if we hash $A_i \# v_i$ (where $\#$ denotes concatenation), and *(c)* keyword-like queries such as "any attribute $= v$" are possible if we hash on v_i. Therefore, we insert each triple (oid, A_i, v_i) three times into the DHT. Though, this data organization produces a certain overhead, there are several advantages of this approach:

- There is no need for a global data dictionary storing schema information – the data are self-describing.
- Triples with similar values are stored at the same peer or at least in the neighborhood which simplifies range queries, joins, and similarity queries.
- There is no fixed schema for a given relation. Users can extend the schema to their needs by simply adding new triples for a given tuple.

However, as a consequence we cannot assume that all users will use exactly the same schema (e.g., attribute names as well as value representations). Therefore, we have to cope with heterogeneities as part of the queries by appropriate similarity operators. Such operators are part of our query language *VQL (Vertical Query Language)* which we introduce in the following only informally.

We chose to base the syntax of VQL on *SPARQL* [12], a query language developed for RDF.But, we only apply the basic syntax, the evaluation of queries in RDF based on graphs has nothing in common with the approach presented here. The basic construct of a query is a `SELECT - WHERE` block similar to the SQL standard. As we do not manage relations in a horizontal manner, there is no need for providing 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, function *dist* is used to express similarity in terms of distance, in our implementation the edit distance for strings and the Euclidean distance for numerical values. Each term starting with a question mark represents a variable. All included expressions are combined conjunctively. Additional clauses `ORDER BY`, `LIMIT` and `OFFSET` are optional.

To understand the concepts behind the syntax of VQL we briefly go through some simple examples. Consider a collection of cars, represented in a database by typical attributes such as name, mileage, price, etc.

- Select name, horsepower (hp) and price of the 5 most powered cars below a price of 50000 (top-N query):

```
SELECT ?n,?h,?p
WHERE { (?o,name,?n) (?o,hp,?h) (?o,price,?p)
        FILTER (?p < 50000) }
ORDER BY ?h DESC LIMIT 5
```

- In contrast to the last query, additionally all corresponding dealers and their addresses (addr) are selected. Moreover, we are only interested in BMW cars:

```
SELECT ?n,?h,?p,?dn,?a
WHERE { (?x,dealer,?d) (?y,dlrid,?d)
        (?x,name,?n) (?x,hp,?h) (?x,price,?p)
        (?y,addr,?a) (?y,name,?dn)
        FILTER (?p < 50000)
        FILTER (dist(?n,'BMW') < 2)}
ORDER BY ?h DESC LIMIT 5
```

So far we looked at some "traditional" operations as well as ranking and similarity on instance level. A powerful aspect of vertical storage is the possibility to express similarity on the schema level which simplifies homogenization tasks. The following example shows the support for similarity operations on the schema level:

- Select all attribute names which have a maximal distance of 2 from 'dlrid', for instance to detect typos. The found dealer objects are joined by similarity on their IDs with car triples in order to list all cars together with corresponding dealers. The result is sorted by distance to 'dlrid' (nearest neighbor ranking):

```
SELECT ?n,?p,?dn,?ad
WHERE { (?d,?a,?id) (?d,name,?dn) (?d,addr,?ad)
        (?o,name,?n) (?o,price,?p)
        (?o,dealer,?cid)
        FILTER (dist(?id,?cid) < 2)
        FILTER (dist(?a,'dlrid') < 3)}
ORDER BY ?a NN 'dlrid'
```

In this work we focus on physical operators, not on issues of query formulation and planning. As exact match queries (see Section 2) and exact match joins [10] are already implemented and evaluated, we concentrate on the implementation of similarity operators. For the remainder of this paper we assume that finally generated query plans are included in messages, which are routed to the processing peers according to the keys the peers are responsible for.

4 Basic Similarity Queries

In this section we will first describe the implementation of a basic similarity operator, which returns all objects with attributes named similar to a search string (schema level) or attribute values similar to a search value in a given attribute (instance level). In the subsequent section we will briefly discuss some advanced similarity operators. Without loss of generality we will focus on queries on single attributes. Queries on multiple attributes can be handled, for instance, by processing separate sub-queries and intersecting the results, or by pre-processing locally materialized intermediate results. Which of these two approaches, or any other, more sophisticated, strategy, is used is a choice depending on cost optimizations, which is part of our ongoing work.

For similarity queries on numerical attributes we map the provided similarity measure to a corresponding interval and process them as range queries. Thus, the applied hash functions have to be prefix preserving. P-Grid supports this and also deals with the possible imbalance problems efficiently (see Section 2), so we will not address this further in the following. For some systems, e.g., Chord, proposals for range queries exist already which can be applied. For most of the other systems which do not support range queries, we have developed a simple range query algorithm, which we cannot present and analyze here for space reasons. The storage scheme facilitates applying exact and similarity operators on instance level using the complete paths from $key(A_i\#v_i)$, as well as on schema level using prefix search on $key(A_i\#v_i)$. For strings, however, the lexicographical order $<_l$ does not reflect the edit distance, e.g., 'a' $<_l$ 'abc' $<_l$ 'd' but dist('a','d') $<$ dist('a','abc').

A naive approach to process string similarity is to send a query to each peer which is responsible for a part of the strings to be compared. The contacted peers then compare the queried string to the data available locally and send matching results back to the peer having initiated the query. As shown in Section 6 this approach does not scale well.

Algorithm 2 String similarity: $Similar(s, a, d, p)$

```
determine q-grams Q from s;
if a == "" then pos = 2; else pos = 3; end if
R = ∅;
for all q ∈ Q do
  T = Retrieve(key(a#q), p);
  for all t ∈ T do
    q' = ξ(t, pos);
    if q' is q-gram ∧|p(q') − p(q)| ≤ d ∧ |l(q') − l(q)| ≤ d then
      oid = ξ(t, 1);
      T' = Retrieve(key(oid), p);
      build complete object o from T';
      for all t' ∈ T' do
        s' = ξ(t', pos);
        if pos == 3 then
          if a == ξ(t', 2) then R = R ∪ {s'#o}; end if
        else
          R = R ∪ {s'#o};
        end if
      end for
    end if
  end for
end for
return (o|s'#o ∈ R ∧ edit(s, s') ≤ d);
```

A more efficient approach are (positional) *q-grams* [7]. This approach is based on the observation that if two strings s_1 and s_2 are in edit distance d, they have to share at least $\max(|s_1|, |s_2|) - 1 - (d-1) \cdot q$ substrings of fixed length q (the q-grams). Additional performance improvements are gained by also taking the starting position of identical q-grams and the length of the corresponding strings into account. We apply the storage scheme as described in Sec-

tion 3, but instead of inserting $key(A_i\#v_i) \rightarrow (oid, A_i, v_i)$ one time, we insert $key(A_i\#q_j^{v_i}) \rightarrow (oid, A_i, q_j^{v_i})$ for each q-gram $q_j^{v_i}$ of v_i, and $key(q_j^{A_i}) \rightarrow (oid, q_j^{A_i}, v_i)$ for each q-gram $q_j^{A_i}$ of A_i. This increases the storage overhead but enables efficient querying on q-grams. With both variants queries are guaranteed to find matching data if it is available.

Algorithm 2 illustrates the basic evaluation of string similarity in our system based on the q-gram approach. p is the peer which currently processes the query. A call to $\xi(t, i), i \in \{1, 2, 3\}$, returns the i-th component of the triple t. If similarity is expressed on instance level then a references the corresponding attribute's name, otherwise a is empty. First (line 1) the search string s is decomposed into a set of q-grams. A natural, but rather expensive choice is to build all overlapping q-grams. As shown in [11] using only a subset of all possible q-grams – a *q-sample* – performs much better but more candidates have to be processed in the final step, resulting in more network traffic. We will compare both strategies in Section 6. For q-sampling we process the search string from left to right and construct $d + 1$ non-overlapping q-grams, starting from each qth position, if s is long enough. Further processing for both variants does not differ: First, all triples matching at least one q-gram are collected; then – as introduced in [7] – *length* filtering and *position* filtering are applied in order to reduce the size of candidate q-grams (line 8: $p(q) :=$ starting position of q in s, $l(q) :=$ length of the corresponding string s); and finally, the complete strings and tuples (in terms of the originating horizontal relations) they belong to are collected, implemented by routing corresponding queries to peers responsible for the collected oids. In the last step we have to determine the final result.

Note that, for simplicity and readability, the pictured algorithm omits two implemented optimization steps. First, queries are delegated from the initiating peer to the q-gram owning peers, which again delegate queries to the oid owning peers, which finally send results back to the initiator. Second, we collect the calls to $Retrieve()$ and contact peers only once using a routing algorithm similar to the shower algorithm in [6]. Applying range queries on attribute prefixes and storing complete strings together with q-grams could potentially improve performance even more.

5 Advanced Similarity Queries

In this section we will sketch two advanced similarity operators: *similarity joins* and *top-N queries*.

Algorithm 3 shows the procedure for processing similarity joins.This operator returns objects from the left set of triples joined with all objects from the right side which are in distance d (if they exist). The semantic of the similarity join is based on the one introduced in [11]. Again, p is the currently processing peer. ln and rn define the attributes on which similarity is processed. For both types of similarity selection, numerical and string based, the basic operators introduced before are used. In addition to extract values from triples, $\xi()$ can be applied on objects here, too. In this first version we process separate similarity selections for each object from the left side, which should be optimized in future variants. Similarity joins on schema level are also possible (processing differs slightly) by leaving rn empty. It is also possible to leave ln empty, though this represents a very expensive operation.

Algorithm 3 Similarity join: $SimJoin(ln, rn, d, p)$

1: $L = Retrieve(key(ln), p)$;
2: $Res = \emptyset$;
3: **for all** $o \in L$ **do**
4: $\quad R = Similar(\xi(o, ln), rn, d, p)$;
5: $\quad Res = Res \cup \{o\#r | r \in R\}$;
6: **end for**
7: return Res;

Another popular query type are rank-aware queries such as top-N. A top-N query delivers the N objects matching a similarity query best, depending on a provided ranking function $rank$. Algorithm 4 shows how we process them. In the following we focus on instance level only. In the current implementation we support ranking functions MIN, MAX and NN. The algorithm as pictured is processed at peer p and works on numerical values of attribute a. We omit illustrating the processing of MIN (analog to MAX). For the different ranking functions the processing differs only in the determination of the range which is queried.

Algorithm 4 Top-N query: $TopN(a, N, rank, v, p)$

1: $c = |\{d \in \delta(p) | key(d) \supseteq key(a)\}|$;
2: determine the size r of the local range of a;
3: $range = N/\frac{c}{r} = N \cdot \frac{r}{c}$;
4: **if** $rank =' MAX'$ **then**
5: $\quad v = \max_p\{d \in \delta(p) | key(d) \supseteq key(a)\} + range + 1$;
6: **end if**
7: $fr, to = Keys(range, rank, v, v)$;
8: $R = \emptyset$;
9: **repeat**
10: $\quad R = R \cup RangeQuery(a, fr, to)$;
11: $\quad range = N/\frac{|R|}{to-fr} = N \cdot \frac{to-fr}{|R|}$;
12: $\quad fr, to = Keys(range, rank, fr, to)$;
13: **until** $|R| \geq N$ '
14: return $Limit(Sort(R, a, rank), N)$;

In lines 1–7 we calculate a first range to query based on the locally provided data density (which is approximately equivalent to the data density on all other peers because of load balancing). This range is calculated such, that it potentially provides all the N queried objects. Thus, we most likely only have to process the first resulting range query: $RangeQuery(a, fr, to)$ starts a range query implemented in P-Grid on range $[fr, to]$ of attribute a. If this does not hold, we adjust $range$ to the data density of the current result and collect oids from an increasing range (lines 10–12) until we have found a set of at least N objects, which is finally sorted and pruned to a size of N (line 14).

The range actually queried is determined by calling $Keys()$, which is illustrated in Algorithm 5.

Algorithm 5 Determine range: $Keys(range, rank, u, v)$

```
1: if rank ==' MAX' then
2:    to = v − range − 1;
3:    fr = to − range;
4: else if rank ==' NN' then
5:    to = v + range/2;
6:    fr = u − range/2;
7: else ...
8: end if
9: return fr, to
```

For the MAX ranking function we start at the upper bound of the attribute's range (if this is not stored locally we can initiate a proper query) and proceed by shifting the search interval downwards. If the ranking function is *NN* we begin in a small symmetric interval around the search key and enlarge it in subsequent steps. For processing top-N queries on strings (only in combination with NN) we have to handle concrete distances instead of interval start and end points.

6 Evaluation

To evaluate our approach, we ran some initial experiments using a simplified simulation. The simulator is written in Java and works on shared memory. The primary performance measures we chose are the number of messages and bandwidth usage, because these are the limiting factors for overlay networks. As the results achieved match our expectations we will include the algorithms in the P-Grid implementation and will run complex experiments on PlanetLab [5].

In the following evaluation we focus on strings, as similarity queries on numerical attributes are based on range queries, which are already implemented in P-Grid and have already been evaluated [6]. We also do not address issues like robustness, fault tolerance, load-balancing, etc., as these have already been evaluated for P-Grid [2, 6]. The experiments were performed on two string data sets: The first one comprises 106704 single words from the English bible, with word lengths from 5 to 14 and an average length of 6.46. The second set consists of 66349 titles of paintings, with lengths from 1 to 132 including spaces. The average length of the titles is 37.08. Due to P-Grid's load-balancing [2] we achieve a reasonable uniform distribution of data items among peers regardless of the actual data distribution.

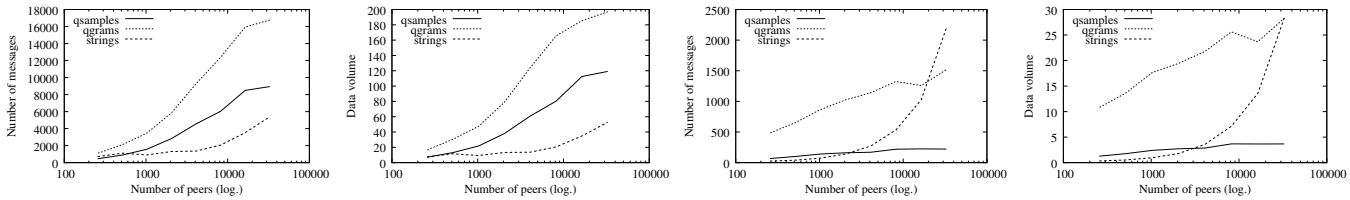
We evaluated two variants of q-gram algorithms, q-gram search and q-sample search, and included the naive approach of querying complete strings for comparison. Comparing these three algorithms we expected the naive approach to perform worst, both in terms of messages and data volume, followed by the q-gram variant, and the q-sample approach as the best performing alternative. We ran tests comprising different numbers of peers in order to evaluate the achieved scalability. In each test we processed a mix of 6 queries initiated 40 times. The set consists of three top-N queries, filtering the $N = 5, 10, 15$ nearest neighbors to a provided search string (up to a maximal distance of 5), and three similarity self-joins over one column. The joins are processed with a maximal join distance of $d = 1, 2, 3$ on the chosen column. In each run we chose the initiating peer as well as the search string (from the set of all strings) of each query randomly and started each of the three methods successively. In Figure 1 we show the averaged results (logarithmic x-scale). *qgram* and *qsample* denote the methods on q-grams, *string* denotes the naive method.

The results gained on the bible words data set shows that the naive string method performs surprisingly well in some cases. However, the figure does not show the enormous effort incurred by comparing the strings at the peers locally, which will result in quite poor query answering times. The reasons for the good performance in terms of messages and data volume are due to the character of the data. Each stored object contains only single words which are fairly short in general. Thus, the advantages of q-grams cannot take effect here. But, the costs of the string approach increase linear in the number of peers and finally it is outperformed by both q-gram methods scaling almost logarithmically, as indicated by the results on bible words and clearly fortified by the results on the titles data. In contrast to the bible words the used titles are fairly long and include spaces, which from our point of view is a more realistic assumption for a wide range of scenarios. Moreover, in real-world P2P systems the number of peers and the number of data is considerably higher (some orders of magnitude), whereby advantages of our approach become obvious. The q-gram methods promise a great scalability, particularly the one based on q-samples. The costs of this method increase scarcely with rising number of peers, which is beyond our expectations. The possible optimizations indicated throughout the description of the method could even improve its performance. The next important step is to evaluate these optimizations and the whole method under the aspects of robustness and self-organization on PlanetLab.

7 Related Work

A number of related works have already been discussed briefly in Section 1. The approach presented in [4] exploits a similar data organization for RDF data but does not address similarity-based queries. Several other approaches exist which consider range queries already as similarity queries. However, we consider here only systems that support already a more complex notion of similarity, such as top-N queries.

LSH forest [3] uses a locality-sensitive hashing (LSH)



(a) Messages (bible words) (b) Data volume (bible words) (c) Messages (painting titles) (d) Data volume (painting titles)

Figure 1. Results of simulation

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 [13] 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 mechanism similar to P-Grid's [2].

Compared to these 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. In conjunction with P-Grid it works efficiently on a wide range of data distributions and network conditions.

8 Conclusions

In this paper we have presented an approach to support similarity queries on structured data in structured overlays to enable public data management applications. Our approach uses a vertically-oriented storage approach based on triples following the idea of RDF and facilitates a richer set of similarity operations than related approaches. Additionally, we define a query language for formulating similarity queries. We used P-Grid as the underlying overlay network but our approach can be generalized to any structured overlay We evaluated the approach regarding performance in terms of number of messages and data volume and showed its efficiency. Though our approach incurs an overhead of storing, publishing and maintaining relations as triples, this overhead should be negligible on modern computers. Due to the pioneering character of the work presented in this paper to support similarity queries in DHTs, a deeper analysis of the resulting efforts and an evaluation of how the approach scales with the number of attributes is still on stage and part of our ongoing work. Currently, we can retain that the overhead of additional overlay messages to reconstruct complete rows of an relation is small as structured overlays offer $O(\log N)$ search complexity and the additional number of messages is linear in the number of attribute columns. As future work we will develop sophisticated processing strategies, do an experimental evaluation on PlanetLab and provide a formal definition of the query language.

References

[1] K. Aberer. Scalable Data Access in P2P Systems Using Unbalanced Search Trees. In *WDAS*, 2002.
[2] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt. Indexing data-oriented overlay networks. In *VLDB*, 2005.
[3] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *WWW*, 2005.
[4] M. Cai and M. Frank. RDFPeers: a scalable distributed RDF repository based on a structured peer-to-peer network. In *WWW*, 2004.
[5] 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), July 2003.
[6] A. Datta, M. Hauswirth, R. Schmidt, R. John, and K. Aberer. Range queries in trie-structured overlays. In *IEEE P2P*, 2005.
[7] 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*, 2001.
[8] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.
[9] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. In *ACM STOC*, 2000.
[10] P. Rösch, K. Sattler, C. von der Weth, and E. Buchmann. Best Effort Query Processing in DHT-based P2P Systems. In *ICDE Workshop NetDB'05*, 2005.
[11] E. Schallehn, I. Geist, and K. Sattler. Supporting Similarity Operations based on Approximate String Matching on the Web. In *CoopIS*, 2004.
[12] SPARQL at W3C. http://www.w3.org/TR/rdf-sparql-query/.
[13] D. A. Tran. Hierarchical Semantic Overlay Approach to P2P Similarity Search. In *USENIX Technical Conference*, 2005.