A DHT-based Infrastructure for Ad-hoc Integration and Querying of Semantic Data*

Marcel Karnstedt, Kai-Uwe Sattler
Faculty of Computer Science and Automation
TU Ilmenau, Germany
<first>.<last>@tu-ilmenau.de
www.tu-ilmenau.de/dbis

Manfred Hauswirth, Brahmananda Sapkota
Digital Enterprise Research Institute (DERI)
NUIG, Galway, Ireland
<first>.<last>@deri.org
www.deri.ie

Roman Schmidt
School for Computer and Communication Sciences
EPFL Lausanne, Switzerland
<first>.<last>@epfl.ch
www.epfl.ch

ABSTRACT

A crucial prerequisite for the deployment and success of Peer-to-Peer data management applications is the availability of metadata in a way that makes it easy to access and combine data from different sources and domains.

In this paper, we argue for a unified and distributed infrastructure providing a repository for semantic data by offering location transparency and advanced query services. After discussing the challenges of such an approach, we present our solution which applies extended SPARQL-like query features for dealing with large and possibly heterogeneous data sets. We focus on the integration into efficient distributed query processing and evaluate our approach in a series of experiments.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—Distributed systems, Query formulation; H.3.5 [**Online Information Services**]: Data Sharing

General Terms

Algorithms, Management, Performance

Keywords

Query processing, Schema correspondences, Semantic interoperability

*The work presented in this paper was (partly) carried out in the frameworks of the EU project TripCom (FP6-IST-4-027324-STP), the Líon project supported by the Science Foundation Ireland under Grant No. SFI/02/CE1/I131 and the EPFL Center for Global Computing supported by the Swiss National Funding Agency OFES as part of the European project NEPOMUK No FP6-027705.

IDEAS08 2008, September 10-12, Coimbra [Portugal]
Editor: Bipin C. DESAI

1. INTRODUCTION

Nowadays, many valuable applications and services particularly on the Web are based on the principle of combining data from different domains and exploiting (semantic) metadata. Typical examples range from information services in life sciences over mashups to Semantic Web applications. In these scenarios, the information, its structure, and its semantics are often controlled by a large number of participants and integration and data management functionalities come into existence through the collaborative efforts of the users.

A crucial task in developing and deploying this kind of applications is therefore to collect, combine, and process data from different domains and sources. Ideally, developers could rely for this task on basic services supporting querying and integrating data similar to services provided by popular search engines. However, building such services for semantic data is more challenging because the data is structured (e.g., RDF data) and appropriate expressive query facilities (e.g., support for SPARQL [15] and beyond) are needed. Furthermore, there are several reasons which argue against a centralized approach as pursued by the leading Web search engines: technical reasons such as scalability and the single point of failure, economical reasons (somebody has to invest in the infrastructure), as well as arguments related to trust and the authority on data.

In this paper, we address these challenges by presenting a fully implemented infrastructure for managing semantically enriched data which comprises a scalable distributed repository for RDF-like data and a distributed query engine for SPARQL-like query languages. Furthermore, we argue for query extensions addressing the problem of heterogeneity of schema and data that is inherent in scenarios where data from different sources and domains is combined.

The remainder of the paper is organized as follows. Based on a discussion of the challenges in developing such an infrastructure in Section 2, we present the architecture in Section 3 and briefly survey related approaches (Section 4). In Section 5 we introduce the basic principles of our query engine, i.e., extended query operators such as similarity operations for dealing with heterogeneous data as well as the main ideas of our distributed processing approach. Next, in Sections 6 and 7 we describe the semantic layer comprising correspondences and accompanying language extensions and their efficient implementation. After showing the results of an experimental evaluation of the overall approach

in Section 8, we conclude the paper and point out to future work.

2. CHALLENGES

The goal of our work is to provide a scalable distributed infrastructure for storing, retrieving, and integrating structured metadata like data for the Semantic Web and similar applications. In contrast to systems such as OceanStore [16], which aim at secure archival storage for a single data source, we aim at integrating data possibly from different sources into a universal, application-independent repository at an Internet scale. While some of the challenges may be similar, there exist several different aspects and others have to be taken into account differently because of disjoint requirements. We overview the key issues for a universal repository in the following and classify the challenges along two major questions:

(1) How to organize and query data in a way also supporting the integration in an ad-hoc or pay-as-you-go manner?
(2) What is necessary to build an efficient, robust, and practical solution?

The first question deals with *data organization* and *query processing* and raises several challenges:

Genericity and flexibility. Because we cannot assume that all users and applications agree to a common schema, a generic and extensible schema is required for structuring data. It should facilitate to add new elements without restructuring or conflicts. This should be accompanied by a schema-independent query language relieving the user of the burden to know relations, classes or element paths. In the Semantic Web the agreed-upon standards for modeling and representing data are RDF/RDF Schema as well as triple stores for data storage which are flexible enough to serve this purpose.

Dealing with heterogeneity. In order to be able to combine data from different domains without forcing all providers to use the same schema, techniques for resolving heterogeneities both on schema level (different names or structures for the same concept) as well as on data level (different representations of the same real-world object) are required. Particularly in large systems and/or loosely-coupled scenarios, combining the relevant data and resolving conflicts should be left to the individual user allowing an on-the-fly or pay-as-you-go integration [14]. However, this needs to be supported by appropriate modeling concepts (e.g., correspondence relationships for schema elements) as well as by explicitly handling schema information as data.

Expressiveness of queries. For the purpose of querying Semantic Web data SPARQL was proposed which resembles many of the core features of a classical database query language for triples and triple patterns. However, in scenarios where users want to combine data from different providers for building new services additional query operators are needed for dealing with unknown schemas and heterogeneities at different levels. For this purpose, IR-style queries are very useful, e.g., keyword searches over all attributes or similarity-based selections and joins. Adding such operations to SPARQL-like query languages would allow to find triples whose properties or objects/literal values are textual similar (e.g., in terms of edit distance) to given strings or even to join them based on a similarity predicate. In addition, a query language for this domain should support querying schema data (attributes, correspondences) as well and treat this as plain data.

The second question touches the *architecture* of such an infrastructure as well as *practicability* issues. Among others the main challenges are:

Distribution and scalability. From a user's point of view the ideal solution would be a central uniform repository where all the metadata is stored. However, this *(1)* raises several problems regarding single-point-of-failure, scalability or resilience against attacks and *(2)* burdens the load or costs of maintaining the service to a single provider. Thus, we argue for a decentralization of data management based on a structured P2P overlay implementing a distributed hash table (DHT). 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 advantage of such systems 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). Furthermore, they provide location transparency: queries can be issued at any peer without knowing the actual placement of the data. An important property of a distributed system is scalability wrt. the number of nodes. For an overlay network based on a DHT this is inherently guaranteed for lookup operations. However, scalability has to be also addressed for more complex query operations as well as particularly for data import/update (e.g., bulk inserts/updates) and more generally maintenance operations.

Efficient query operators. Using a distributed infrastructure for data management opens several alternatives for building query services: by data shipping meaning to fetch data from the providers (e.g., using the lookup service of the underlying storage structure) and process the query at the requesters side – or by query shipping meaning to send (parts of) the query to the nodes where the data is stored. The latter is often the better choice but – depending on the partitioning of the data – requires distributed implementations of the query operators. These implementations should come with worst-case guarantees (e.g., $O(\log n)$ for DHTs). Furthermore, they should exploit the features of the underlying infrastructure, e.g., for DHTs hash-based placement and indexing facilities or topology-aware routing and multicasting for range queries. Finally, processing more complex queries, which typically result in several equivalent execution plans, should involve cost-based and adaptive query optimization considering the dynamicity of the whole network and the autonomy of the individual nodes.

Robustness and availability. A distributed storage has to be robust and reliable, which basically means to be resilient against node and link failures. This has to be addressed by maintaining redundant links (as already provided by DHTs), but also by replicas of data. Data replication raises several further issues, such as the required number of replicas in order to guarantee a degree of availability or the strategies used for update propagation in a decentralized environment. Moreover, the existence of replicas allows the system to choose among different nodes for retrieving data based on the current load of nodes, but requires distributed monitoring of load in a dynamic environment.

Privacy and trust. In a DHT-based overlay network, where data is not stored on a provider's site, there exists

a strong requirement to prevent malicious behavior of nodes (e.g., modifying locally stored data). Thus, privacy of the hosted data as well as trusting peers on the returned result of a query are further important challenges.

In the following sections, we sketch our approach to meet most of these challenges and briefly discuss ways to address the remaining ones. The focus of the work in hand lies on the the technical aspects of integrating semantic correspondences into the flow of query processing. For convenience, we briefly present fundamentals of query processing and specialized operators tailor-made for large scale heterogeneous data repositories. Further, we use a SPARQL-like query language, providing valuable extensions for integrating semantic correspondences into query formulation, as a basis for the technical discussion. The UniStore [13] system provides the needed data management and query processing capabilities.

3. ARCHITECTURE

The scalable infrastructure for the integration and querying of semantic data is implemented in three layers as shown in Figure 1. The *semantic layer* enables virtual grouping of triples as well as creating mappings between these triples. The *distribution layer* provides support for transparently distributing data, query as well as index structures over the network whereas the *distributed data layer* offers storage, indexing and querying of triples either locally or in the network.

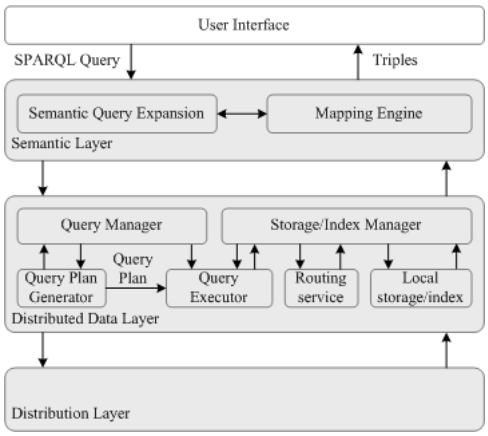

Figure 1: Distributed semantic storage

As mentioned in the introduction, we aim to provide a scalable distributed infrastructure for storing, retrieving and integrating structured metadata like data for the Semantic Web. In this regard, a natural way of supporting these features, to some extent, is to exploit scalability, location transparency, logarithmic search complexity as well as guarantees offered by a DHT infrastructure [1]. In our implementation, the P-Grid DHT is used in the distribution layer because of its support for scalability as well as robustness, as required by our infrastructure. In the distributed data layer we implement a flexible triple storage and build several indexes using the capabilities of the distribution layer. Query processing is provided by a set of database-like operators working on these indexes, which completely rely on the DHT features of the distribution layer.

It is natural to anticipate that different (groups of) people or users often organize their data using their own schema instead of using a common one. In addition, data published in the Web by one use user could be useful to another user. Therefore, mappings are necessary for establishing semantic connections between these schemas in order to share data as well as to integrate data sources. However, creation of such mappings is easier and effective if schemas are defined clearly, such that the names and structures used for representing a concepts can be identified. To support this activity, technologies from the Semantic Web are exploited in the semantic layer in order to represent different concepts and the relations between them.

In order to support the formulation and processing of queries, we propose a structured query language that extends SPARQL-like query features for dealing with large and possibly heterogeneous data sets. These queries first need to be routed to the places where the semantic correlations can be resolved. This integrates straight forward into query processing, because semantic data is stored in the same triple format as plain data. Afterwards, the queries need to be routed to the peers where they can possibly be answered. The main principle of the operators used for this is to use the DHT hashing features on different (combinations of) parts of the managed triples – may they contain semantic data or plain data.

The described architecture is implemented in the UniStore system [13] using Java. UniStore comes with a light-weight, portable client. It is developed to build distributed universal storage systems on large scale. The implementation is designed modularly to be able to replace the DHT used in the distribution layer. But, currently, it makes explicit use of a variety of efficient functionalities provided by P-Grid, such as efficient range querying. We present the main facts about its underlying data model and its distributed query engine in Section 5.

4. RELATED WORK

GridVine [2, 6] is a peer data management infrastructure addressing both scalability and semantic heterogeneity. Scalability is addressed by peers organized in a structured overlay network forming the physical layer, in which data, schemas and schema mappings are stored. Semantic interoperability is achieved trough a purely decentralized and self-organized process of pair-wise schema mappings and query reformulation, forming a semantic mediation layer on top and independent of the physical layer. The semantic layer enables peers to share data in the overlay network according to local schemas interlinked by user-defined schema mappings. These schema mappings are then used for automatic query reformulation allowing queries to traverse a sequence of schemas at the mediation layer and retrieve all relevant results, irrespective of their schemas. GridVine offers a recursive and an iterative gossiping approach, i.e., query reformulation approach. In iterative gossiping, the peer issuing the original query is responsible for retrieving all mappings and reformulating all queries by itself iteratively. In recursive gossiping, the reformulation process is iteratively delegated to those peers receiving reformulated queries. Recursive gossiping performs systematically better because it distributes the reformulation load among peers.

GridVine shares most of the aims and features of our approach, e.g., both systems use the same underlying overlay

network. The differences relate to the challenges expressiveness of queries and dealing with heterogeneity. GridVine implements triple pattern queries with support for conjunctive and disjunctive queries realized by distributed joins across the network. They do not apply the idea of cost-based, DB-like query processing over multiple indexes. We support similarity-enriched SPARQL-like queries with in-network query execution realized by query plans executed in parallel. On the contrary, we currently do not implement something similar to GridVine's verification of schema mappings [5] to identify incorrect schema translations. Though, both systems are able to handle data and schema heterogeneity to the same extent. GridVine also deals with transitivity and cycles in schema correspondences in more detail than we do up to this point.

Piazza [7] is another peer data management system dealing with scalability and data heterogeneity. It uses a mapping language for defining mappings between sets of XML source nodes with different document structures (including those with XML serializations of RDF). The system uses the transitive closure of mappings to answer queries. It is able to follow mappings in forward and reverse directions and can both, remove and reconstruct XML document structure. Unlike our approach, Piazza uses central indexing. An extension to use an underlying DHT only seems to be planned for the future, a key requirement for Internet-scale systems.

RDFPeers [4] presents a scalable and distributed RDF triple repository for storing, indexing and querying individual RDF statements. RDFPeers self-organize into a multi-attribute addressable network (MAAN), which extends the Chord DHT to efficiently answer multi-attribute and range queries. The system's query processing capabilities are very similar to the ones of GridVine. It supports triple pattern queries, disjunctive and range queries, and conjunctive multi-predicate queries using RDQL. In contrast to our approach and GridVine, query resolution is done locally and iteratively. Further, RDFPeers also does not address the problem of schema heterogeneity and is therefore not as generic and flexible as GridVine or us.

PIER [8, 9] is an Internet-scale query engine built on top of the distributed hash table CAN. The system aims at distribution and scalability challenges as our approach, applying similar DB-like query processing concepts. But, PIER does not consider the problem of heterogeneity. They assume that agreement on a global schema is feasible and desirable.

5. FUNDAMENTALS OF QUERY PROCESSING

The data model underlying the distributed data layer is based on the ideas of RDF and a universal relation model. Data is organized as a set of triples. From a database point of view this is modeled using a single relation with the attributes object URI (also called object id, OID for short), property A, and value v. Note, that a tuple of an arbitrary relation of a schema $R(A_1, \ldots, A_n)$ can be represented as a set of triples `<oid,` A_1`,` v_1`>` ... `<oid,` A_n`,` v_n`>`. Like this, the (virtual) universal relation is vertically partitioned.

Based on this, we use an own query language on the notion of SPARQL. However, our main goal is not to build simply yet another SPARQL processor, instead we are interested in advanced query concepts. Thus, in the following we focus on the extended features and assume the reader's familiarity with the basic concepts of SPARQL and similar languages.

Queries are transformed into logical query plans. Each logical operator has a variety of physical implementations. These physical operators differ in the actual processing and routing strategies applied and in the indexes used. Logical operators are either replaced by physical counterparts before issuing queries (static processing), or during query run time (adaptive processing).

In order to process queries efficiently in a large-scale distributed environment, sophisticated techniques are required. First of all, the lookup mechanism of the underlying DHT is exploited by building multiple indexes: one indexing each triple on the OID, one on the concatenation of the attribute name and value ($A_i \# v_i$) for prefix searches, and a third on the value v_i itself. Operators are routed by computing hash keys on corresponding parts of triples and using the DHT routing facility. This enables, for instance, search based on the unique OID, queries of the form $A_i \geq v_i$, and using v_i as the key for querying on an arbitrary attribute. Second, we implemented a q-gram-based approach for supporting similarity operations. This 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). For this purpose, the q-grams of each property name and (string) value are used as keys in the indexing scheme, too. Based on this, special implementations of the query operators for similarity selection, similarity join as well as skyline processing are provided. These operators are described detailedly in [12, 10].

We use distributed implementations of all query operators together with a query shipping approach, which works as follows. Each query is translated into a query plan, wrapped into a message and routed to the affected nodes, which are identified using the hash function(s) of the underlying DHT. Once a responsible node receives a query plan, it processes (parts of) the plan locally, inserts the (partial) result into the plan and forwards it. This is repeated until the final result is received at the query's initiator. If multiple peers are responsible for a queried key (range), the corresponding operator can be processed in parallel. This is called *intra-operator* parallel. If operators from different branches of a query plan are processed in parallel, this is called *inter-operator* parallel. Query processing typically results in multiple plans for a single query. These plans are routed independently trough the network, processed in parallel (if not too expensive), and synchronized at the initiator or another designated peer. In this paper, we focus on how to integrate the querying and resolving of schema correspondences into this flow of stateless query processing. In the following, we briefly present some specific extensions tailor-made for large-scale heterogeneous data sets.

The first language extension is a similarity-based function `edist(`x`,` y`)` for the `FILTER` clause, that calculates the popular edit distance (also known as Levensthein distance) of two strings x and y. The edit distance is the minimal number of required character edit operations (insert, delete, modify an character) to change the string x into y. A similarity predicate can be used both for values as well as schema elements (properties). This is illustrated in the following example query, which returns all triples having a property with a name similar to "city" and a value similar to "Munich":

```
SELECT ?o ?a ?v
```

```
WHERE { ?o ?a ?v .
  FILTER(edist(?a, "city") < 2) .
  FILTER(edist(?v, "Munich") < 2) }
```

Furthermore, similarity predicates can be also used as join conditions in order to combine tuples which have similar values in the join attribute. This also allows for similarity predicates on schema level.

The second extension is a top-k query operator. Using the `ORDER BY` clause, the triples can be sorted wrt. a given criteria and the resulting list is truncated by a specified `LIMIT` value. This also works with a similarity function, which allows to formulate a k-nearest neighbor predicate, e.g., looking for the ten most similar values to "Munich":

```
SELECT ?o ?v
WHERE { ?o address:city ?v }
ORDER BY DESC(edist(?v, "Munich")) LIMIT 10
```

Finally, we also support skyline queries by introducing a special query operator and an accompanying `SKYLINE` clause. A well-known definition of the skyline is based on the *dominance* relation [3]: given two or more ranking functions, the skyline of a data set is formed by all points of the set not dominated by any other point, i.e., a skyline member is ranked higher than a non-member in at least one dimension of the skyline. In mathematics and economy this set is known as the *Pareto optimum*. Members of a skyline represent best possible trade-offs between all ranking goals, which is, for instance, extremely helpful in decision making tasks. A popular example is to ask for all hotels which are preferably close to the beach (min(distance)) *and* preferably cheap (min(price)):

```
SELECT ?o
WHERE { ?o hotel:distance ?d .
  hotel:price ?p }
SKYLINE MIN(?d) MIN(?p)
```

The introduced extensions and corresponding operators are already implemented in UniStore. Now, the task is to extend the query processing and query formulation strategies to support ad-hoc integration of semantic correspondences. For instance, the skyline query from above should also return results from providers using different data schemas, like `accommodation:rate` for price information. Preferably, the user issuing such a query should not be expected to provide the knowledge needed for this expansion.

6. SEMANTICALLY EXPANDED QUERYING

6.1 Representing Correspondences

We start our considerations from the fundamental correspondence type *attribute equivalence* $A \cong B$, which means attribute A of schema one represents the same information as attribute B in schema two. This basic relation can be found very often in the scenarios considered, e.g., one user or service supporting attribute "birth", a second one providing the same information in attribute "dob". This is illustrated in a small example about city data in Figure 2. Two relations (i.e., a set of attributes that belong to each other) are sketched and two equivalence mappings between attributes are indicated. This can be enriched by other correspondences like subclass and also be extended to relations themselves, respectively concepts they represent.

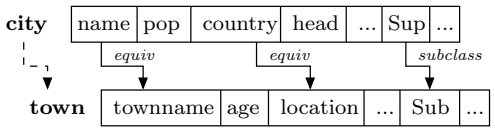

Figure 2: Example schema correspondences

Such mapping information can be stored using the proposed storage model in a single triple, e.g., `<A, map:equiv, B>`. Of course, correspondences between attributes may be much more complex, spanning from algebraic operations (e.g., currency translation) over attribute combinations (concatenation, splitting, ...) to complex filter statements (e.g., when defining complex views). We focus on basic correspondences, enriched by the optional involvement of additional data, for instance from an ontology. In Section 7 we will show how both, basic mappings as well as usage of an ontology, can be integrated easily and well-fitting into query formulation and processing, emphasizing the light-weight and stateless character of the proposed approach.

There has been much work on representing ontologies in a triple format, e.g., RDF-like. Several of these approaches qualify for representing ontology information in our framework. One popular approach used in many applications is OWL. As the actual expressiveness and applicability of the used form of representation is not the focus of this work, throughout the paper we will restrict ourselves to some basic, intuitive and self-explanatory relations, like *subclass*, *superclass*, etc. This can be easily extended to relations between multiple ontologies and to vocabularies like OWL. Similar, an attribute is not bound to only one ontology or only one relation. A triple like `<A, ont:belongs, C>` represents a kind of entry point into more complex ontologies.

The approach reveals several advantages:

- additional mapping information can be added with ease
- easy integration into storage and processing model
- information management and discovery is distributed

Of course, this comes along with increased query overhead (the mapping information must be queried as well) and increased storage overhead. We regard these costs as implicit and unavoidable costs of realizing semantic interoperability. Next, we will discuss how to integrate such schema correspondences into query formulation.

6.2 Query Formulation

This section clarifies the basic approach and highlights the tight and easy integration of schema correspondences into query formulation. Details of actual query processing are presented in Section 7. We base our discussion on a modified version of the well known RDF query language SPARQL. Everything we propose here can be easily integrated into SPARQL itself, which was not done by ourselves in order to focus on the other challenging aspects.

We start from a simple example about services sharing personal and geographical data. Imagine the triples

```
<03, person:hometown, mainz>
<x23, people:livesin, london>
```

as a part of two sets of user data about people. Both attributes represent the name of the city a person lives in. The corresponding information can be stored in a triple

<person:hometown, map:equiv, people:livesin>

We use prefixes in the attribute names, which is not mandatory. As this eases some queries and the identification of relations, but does not restrict the generality of our approach, we use these prefixes as a kind of relation identifiers.

Imagine the following example query looking for all persons from German cities, which involves an URI join in one relation (relation `person`, one tuple identified by URI `?o`) and a join between this relation and a next one (on attributes `person:hometown` and `de:city`):

```
SELECT ?name
WHERE { ?o person:name ?name .
  ?o person:hometown ?city .
  ?o2 de:city ?city }
```

This assumes that relation `de` only stores German cities, which is a kind of implicit semantic knowledge. This fact does not hamper the illustration of the general idea, rather it helps to understand specific problems. Moreover, it shows that also such implicit semantics are supported.

Now, the correspondence information stored in the system must be used in order to rewrite the query and to expand the result set accordingly. Usually, the question which correspondences when to apply is evaluated on a complex set of rules using inference techniques. In a distributed and decentralized environment this can be very expensive. The focus of this work is on the actual (and light-weight) integration of mapping information into distributed query processing. Thus, we use such mappings as a special kind of inferences in order to implement a basic rewriting. Knowledge about several basic rules mandatory for this task is assumed. Details about special challenges on the logical level, like resolving transitivity and reflexivity, different kinds and representations of correspondence information, or algorithms for finding correspondences are subject of other works. Insights gives, for instance, the GridVine system [2, 6].

An expanded query will have a similar shape like this:

```
SELECT ?name
WHERE { ?o person:name ?name .
    ?o person:hometown ?city .
    ?o2 de:city ?city
  UNION
    ?o people:name ?name .
    ?o people:livesin ?city .
    ?o2 de:city ?city
  UNION
    people:name map:equiv ?a1 .
    ?o ?a1 ?name .
    people:livesin map:equiv ?a2 .
    ?o ?a2 ?city .
    ?o2 de:city ?city }
```

Each part of the union operation symbolizes one set of resolved mappings, each corresponding to a specific semantic relation. The second part shows how simple mappings like the one from above are resolved, the third one how the identification of more complex correspondences can be integrated into query formulation (here we look for transitive mappings). Integrating additional ontology information is straight forward. Note that a triple pattern like `<?o, ?attr, ?v>` usually results in materializing all triples available in the system, as none of the parts is bounded. We allow such patterns only if a corresponding bound results from other triple patterns (as in the example), which can be detected at query translation time.

Like this, semantic expansion is integrated directly into query formulation and, as we will show in the next sections, into query processing. But, this requires explicit semantic as well as expert knowledge in order to build these queries. Moreover, this information must be available locally. For a standard user, issuing queries from the user level (i.e., without the need for explicit knowledge about semantic information, its structure and the actual integration into querying) is much more preferable. For this, we propose to use a kind of semi-automatic query rewriting. The query only specifies whether query expansion shall be triggered or not, optionally on which attributes. This can be extended in order to provide additional information, like a maximal level of transitivity or an instruction indicating to involve ontology data. A conceivable form could be like:

```
EXPAND      person:* 1
ONTEXPAND sub
SELECT ?name
WHERE { ... }
```

This means that all correspondences for attributes with prefix `person` are resolved, including the subset relationship between ontology concepts. This information is transformed into corresponding query operators, shipped to where the semantic data is found, resolved, and afterwards processed on the mapped attributes. The implementation of this is discussed in the next section.

7. SEMANTICALLY EXPANDED QUERY PROCESSING

Up to here, we have motivated the need for an integrated and ad-hoc querying of heterogeneous semantic data. Further, we introduced a way of issuing corresponding queries, on a direct level requiring sophisticated experience, as well as on a more user-friendly, indirect level. But, this only helps to achieve the aspired challenging aims if it comes along with an appropriate efficient query processing. The specific requirements are: *(i)* to prevent a negative effect of the query expansion on the performance of the original, unexpanded query. On technical level this means to prevent increasing response times for results belonging to this original query. *(ii)* load injected into the system by additionally querying mapping data has to be minimized. In the distributed environments we propose, this mainly reflects in the number of additional messages generated and the amount of additional bandwidth used.

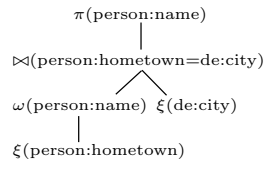

Figure 3: Unexpanded query

These objectives are achieved by adhering to the principles of query processing introduced in Section 5. Particularly, se-

Symbol	Operator	Description
ξ(A)	Extract A	Extracts all triples for attribute A
ξ(A←B*)	Extract prefix B	Extracts all triples for attributes prefixed by B – attributes are later referenced by A
⋈(A=B)	Equi-Join A=B	Join all input triples on predicate A=B
ω(A)	Materialize A	Extracts all triples for attribute A and combine them with corresponding input triples - join on object ID
π(S)	Project S	Project values for all attributes from set S
∪	Union	Combine both sets of triples
map(A)	Map A	Apply semantic expansion on attribute A (adaptively)
β(A←B)	Rename A←B	Rename attribute B to A (also used on multiple attributes)
λ(O,A)	Lookup O	Hash-Lookup on OID O, corresponding attributes are later referenced by A

Table 1: Used query plan operators

mantic expansion is implemented by integrating according sub-queries into the original query plans. Like this, they are tightly integrated into the process of query planning. Of course, mapping information could be queried in analogy using separate queries. After collecting all needed information, the original query could be expanded at the initiating peer. This is contrary to the aspired tight integration into stateless query processing.

As introduced, the query plans are copied and processed in parallel, according query plan operators can be processed in parallel as well. We rely on the principle of parallelism as one key factor to achieve scalability in dynamic environments. We will illustrate the details of this using a database-like notation of query plans. A node in these graphs represents a single operator, processing data from the down to the top along the edges as soon as it is available. For the different operations on data level, we use different operator symbols shown in Table 1. Each operator can process each of the three parts, or a combination of them, of input triples. During query formulation this is represented using according variables in the used triple patterns, see Section 6. We stay to this principle, but rather than using the variable names, for readability we use the attribute identifier to illustrate how triples are identified and processed. We use the following notation: When referring to the content of an attribute A, e.g., for extracting all triples containing values of A, we use A in the operator notation. In predicates, e.g., used in joins to combine triples, using A refers to the instance value of the attribute A. As queries on schema level are supported as well, we refer to the actual name of the attribute A by using $\overline{A}$ (second part of a triple). $\hat{A}$ refers to the identifier of the triple (URI or OID). For instance, applying ξ(name) extracts triples like `<t123,name,Karnstedt>`, where references to the attribute "name" in the query plan refer to $\hat{name}$=t123, $\overline{name}$=name, and $name$=Karnstedt.

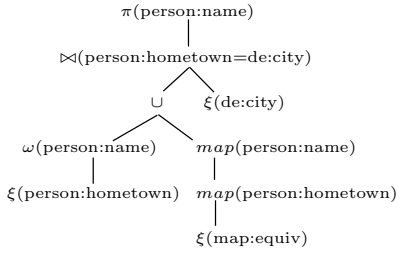

Figure 4: Query with mappings for expansion

Query plans are usually processed in post-order. Figure 3 shows the query plan corresponding to the unexpanded user query introduced in Section 6. First, a separate copy of the plan is sent to each node responsible for attribute `person:hometown`. These nodes replace the ξ-operator with according triple data, and forward each copy to all peers responsible for `person:name` (copying the plans again, if needed). These nodes join triples for both attributes on shared identifiers (URI) and again forward copies of the query plans. Finally, all nodes hosting information for attribute `de:city` receive all resulting copies and insert local data. As all needed data is available now, the remaining query plan operators are processed as well and final answers are returned to the query initiator. This means triples from both sub-trees are joined on a shared value for the city attribute, i.e., triples without an appropriate partner are filtered out. Finally, the values of attribute `name` are projected from the remaining triples and replies are sent. The separate plans "travel" independently through the network and deliver result data iteratively back to the initiating side. We also support estimating the number of answers generated by this without the need for global knowledge at the initiatior's side ([11]). The benefits of this are: query processing is highly parallelized and stateless, whilst robust in dynamic environments, but still predictable. To the user as well as each involved node this represents a comfortable "fire-and-forget" querying.

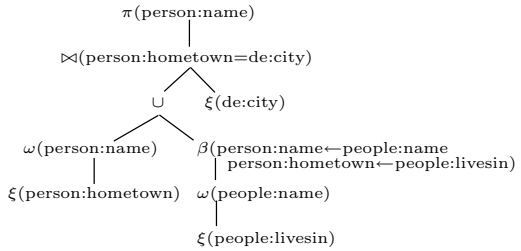

Figure 5: Expanded query

If mapping information is directly integrated into a query using according triple patterns, these query parts result in separate sub-trees of the corresponding query plan. Semi-automatic expansion is implemented following the idea of adaptive query processing. In this case, the user is not required to formulate complex queries on mappings and ontologies. For this, we introduce a *map* operator, which fits into the notion of query plan operators and mutating query plans. If the need for semantic expansion is indicated in an initiated query, corresponding parts of the query plans are cloned and added as new children of a union operator. In order to achieve the first requirement introduced above, to not effect the response time of first replies, both sides of the union operator are not processed in post-order. Rather, they are processed in parallel (we call this intra-operator parallel). As the actual union of both input sets is data in-

dependent, this can be done during query processing on arbitrary intermediate nodes, as well as, preferably, by simply combining replies at the initiator side. Operators referring to attributes to be mapped are temporarily substituted by *map* operators, as shown in Figure 4. Additionally, subquery plans are inserted in order to materialize the information needed to resolve the mappings. In the case of simple equivalences, this may be just one operator extracting all triples on the `map:equiv` attribute. On each of the nodes responsible for this attribute, semantic expansion is finalized by (re-)inserting the appropriate operators (like ξ and ω in the example), but this time using the identified mapped attributes. Due to the usage of according renaming operators, the map operator can be implemented independent from other operators and its actual position in the query plan. The resulting expanded query plan is shown in Figure 5. A renaming operator β is inserted automatically every time, even if it is not mandatory. Subsequent operators can access mapped attributes (from `people`) with their originally intended names (from `person`). This provides flexibility in plan processing, planning and optimization. The expanded query plan is the result of an iterative process: only after all necessary information is gathered the plan can be finally rewritten. If the corresponding semantic data is distributed over several peers (the more complex the sub-query is, the more peers will be involved), this will happen at the last peer responsible for some of the metadata. If parallelism is used, there must be an according peer for synchronizing the parallel sub-plans.

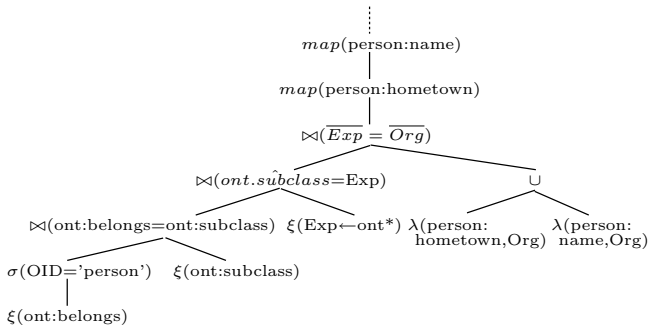

Figure 6: Expansion using ontology

As an extension, we can also query complex ontologies using the same approach of parallel and stateless query processing. This reflects in replacing the extraction for attribute `map:equiv` in Figure 4 by a more complex sub-query plan, as shown in Figure 6. For readability, we only show the part of the query plan reflecting the mapping process. This would extract subclass information from an ontology-like data as:

```
<person , ont:belongs , people>
<person:name, ont:attr:id , people>
<person:hometown, ont:attr:city , people>
<celebrities , ont:subclass , people>
<whoiswho , ont:belongs , celebrities>
<whoiswho:givenName , ont:attr:id ,
    celebrities>
<whoiswho:city , ont:attr:city , celebrities>
```

First, information on the ontology attributes `belongs` and `subclass` are extracted and joined accordingly. Afterwards, all attributes belonging to ontology concepts are extracted, using a prefix query on the commonly used prefix "ont". This could be extended using multiple different prefixes or similarity queries on schema level. Further, re-usage of shared expressions can help to optimize query processing. A join combines the so extracted attribute candidates and the concepts found before. Here, the usage of a prefix query avoids the problem of processing `<?o, ?a, ?v>`, which would materialize all triples in the system. An alternative is to use a nested loop approach in the join operator (the left side is already materialized) or to use adaptive operators again. Finally, to materialize all information needed for the actual mapping, a last join on schema level combines all candidates with all ontology triples for the attributes that shall be mapped (extracted using a lookup on corresponding OIDs). After that, the mapping operators can be processed accordingly. Note that this is only one of various possible query plans.

Using this parallelized approach, result data from the original query is materialized as fast and efficient as before. Semantic expansion results in an expanded query result, which usually will result in an extended query time as well. We will further argue on this in Section 8. The advantages are: processing is still highly stateless and thus scalable, efficient due high grade of parallelization, and of course the expansion of the result set without any interaction required from the user. The generic nature of the approach, arbitrarily combining different triple parts, flexibly supports different representations. The exact overhead will depend on the kind an complexity of extracted semantic data. The query processing and planning facilities allow to process arbitrary parts of a query plan in parallel. For instance, each join in Figure 6 could be processed in parallel similar to the union operator. The actual decision on this reflects a trade-off between message overhead, bandwidth consumption and robustness, as well as response time. Details can be found in prior works, e.g., [12]. The fact that user queries can easily end in very poor performing query plans (challenging for each query planner) highlights another important advantage of indirect query expansion in combination with adaptive query processing, because it is cost-based and solely due to the query engine.

8. EVALUATION

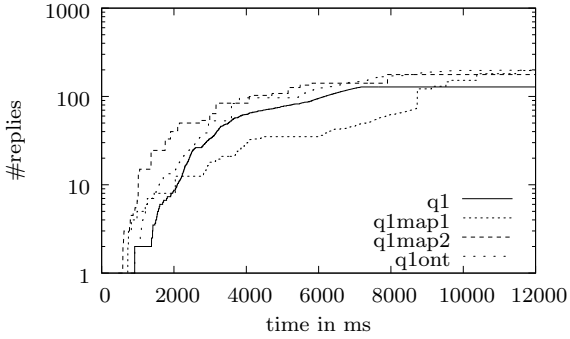

Figure 7: Query q1 expanded on 70 nodes

The objective of the following evaluation is to show that we can achieve our main requirement, which is efficiency in query processing. Specific operators tailor-made for building a repository on heterogeneous data, like ranking or similarity

operators, have been evaluated before [10, 12]. Here we show that we first meet our main requirement, not to influence performance of an unexpanded query. Further, the following results document that query expansion is processed at a very low overhead and perfectly integrated into parallelized query processing.

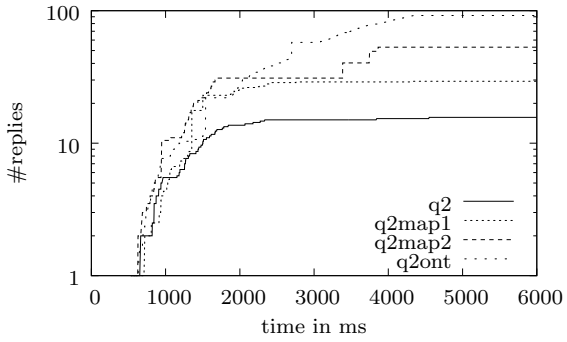

Figure 8: Query q2 expanded on 70 nodes

We ran our tests on a network of up to 70 instances in our local environment. The data we used has geographical nature. We extracted triple data from DBpedia[1] and, for having some realistic background as well as well-defined query results, we combined this with some input from the Mondial database[2] and some local ontology data. From this set of about 30000 triples, we inserted about 7000 randomly chosen triples into the system. On these triples, we built two indexes using the infrastructure, one on the attribute names and one on the identifiers (URIs). Thus, we handled about 14000 triples in our system. In the following, we present a selected sub-set of all results gathered. In each test, we ran a set of chosen queries, each initiated 5 times by randomly chosen peers. Among the set of executed queries, we focus on two very representative ones. Other results look similar.

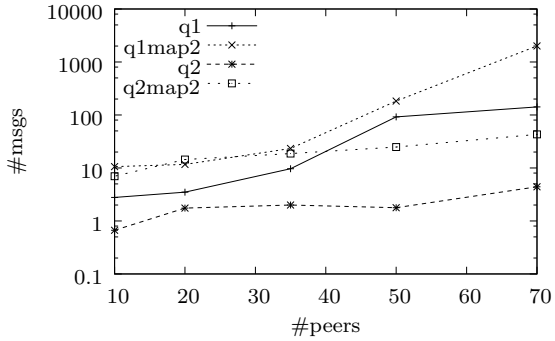

Figure 9: Message overhead

The first results show the performance of queries on the largest network size of 70 nodes. In Figures 7 and 8 we plot the number of replies received over time. We plot the number of replies rather than the result size, because this size depends too much on the query and data selectivity. The result size increases nearly in parallel to the number of replies, but showing more leaps. The number of replies (including empty replies) gives a better insight on the actual performance. We show the results for two queries. The first, $q1$ is equally shaped as the user query introduced in Section 7. The difference is that we inserted an additional filter operator, use similarity joins in order to achieve meaningful result sizes on the heterogeneous data and apply a final materialize operation below the root of the query plan. The second query $q2$ looks similar, but includes a second similarity join. Each query was run separately, and afterwards expanded by one mapping ($q * map1$) and two mappings ($q * map2$). Further, we used the ontology query shown in Section 7 for expanding the original queries, represented by $q * ont$. As proposed, queries were strictly parallelized. Both plots clearly show that running an expanded query does not influence response time negatively – rather, in some cases the results of expanded queries arrive earlier. All in all, the plot for the number of replies shows a similar shape for all queries, all the same how much parallelism is involved or how complex the query is (e.g., $q * ont$).

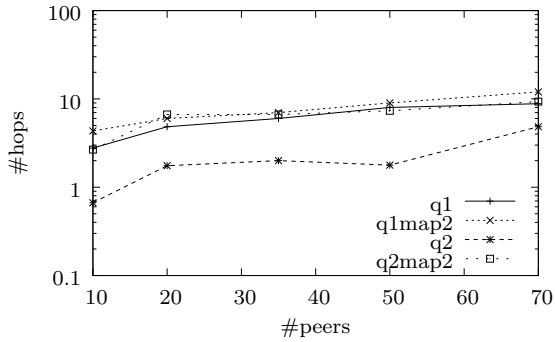

Figure 10: Hop overhead

This is due to the parallel processing approach, which comes at the costs of messages. This is illustrated in Figure 9 (note the logarithmic scale). The number of messages increases almost exponentially with the network size for some of the queries. For others, the overhead is constant. This mainly depends on the number of intra-operator parallel processing stages that a query plan contains. Bandwidth consumption raises almost in the same scale like the number of messages. The though low response times are reflected by the number of hops shown in Figure 10.

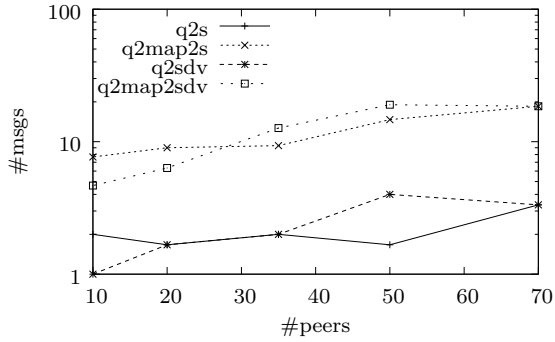

Figure 11: Varied parallelism

There may be a point where such an amount of generated messages results in performance leaks. The more parallel operators in a post-order plan, the higher the number of

[1]http://www.dbpedia.org
[2]http://www.dbis.informatik.uni-goettingen.de/Mondial/

messages. This can be avoided by increasing inter-operator parallelism (process more sub-trees in parallel) or by using alternative operators in specific situations (this decision is due to a sophisticated optimizer). We show this for $q2$ in Figure 11. $q2 * s$ are the same queries as before, but all materialize operations are run using a sequential operator, which decreases parallelism. The number of generated messages clearly drops. Finally, we show that the size of the inserted data has very low impact on the query performance: queries $q2 * sdv$ are the same as $q2 * s$, but were run using increasing data volume with increasing network size (each node inserted the same amount of triples, up to the size of 14000 handled triples). Response times for all queries are very satisfactory, as Figure 12 shows. We plot the time averaged over all results for some selected queries. The time for the last result increases linearly with the network size, as expected. The time for the first result behaves very similar like the average time plotted – which is backed by the plots in Figures 7 and 8. Most of all results and replies are received in the first seconds after issuing a query. With respect to the network size, response times fortunately even decrease.

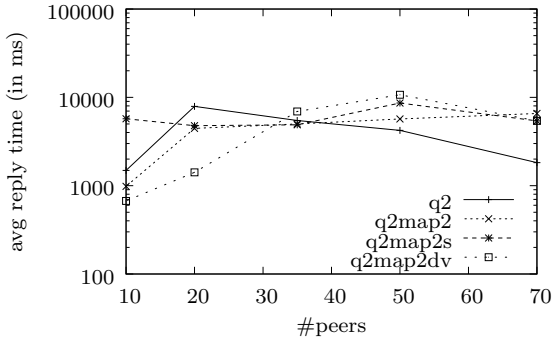

Figure 12: Varied data volume

All in all, the presented results only give an insight in actual performance of query processing. But, they absolutely justify the statement that the proposed approach fits well into the implemented query processing strategies. With this foundation, querying a heterogeneous, semantically enriched data storage emerges as simple and efficient as querying the Web itself, but also as powerful as querying database systems. This all comes with low effort and no expert user knowledge or intervention needed at all.

9. CONCLUSION

In this paper, we discussed several of the main challenges of developing a unified and distributed infrastructure for building an advanced semantic data repository. We indicated how schema correspondences, needed for querying heterogeneous data, can be integrated into query formulation. Then, we focused on the efficiency and modularity of integrating the process of query expansion into distributed query processing. Such semantic interoperability can be achieved unproblematically with low additional effort on large scale, if integrated into DB-like distributed storage systems like UniStore. In future work, we will extend the set of supported semantic correlations and inferences, as well as further evaluate the proposed infrastructure in the context of the Semantic Web and similar applications.

10. REFERENCES

[1] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Punceva, R. Schmidt, and J. Wu. Advanced peer-to-peer networking: The P-Grid System and its Applications. *PIKJ, Spec. Issue on P2P Systems*, 26, 2003.

[2] K. Aberer, P. Cudré-Mauroux, M. Hauswirth, and T. V. Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In *Int. Semantic Web Conference*, pages 107–121, 2004.

[3] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE'01*, pages 421–432, 2001.

[4] 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.

[5] P. Cudré-Mauroux, K. Aberer, and A. Feher. Probabilistic message passing in peer data management systems. In *ICDE '06*, page 41, 2006.

[6] P. Cudré-Mauroux, S. Agarwal, and K. Aberer. GridVine: an Infrastructure for Peer Information Management. *IEEE Internet Computing*, 11(5):864–875, 2007.

[7] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW '03*, pages 556–567, New York, NY, USA, 2003. ACM.

[8] 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.

[9] 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.

[10] M. Karnstedt, J. Müller, and K. Sattler. Cost-Aware Skyline Queries in Structured Overlays. In *ICDE Workshop on Ranking in Databases (DBRank'07), Istanbul, Turkey*, pages 285–288, 2007.

[11] M. Karnstedt, K. Sattler, M. Haß, M. Hauswirth, B. Sapkota, and R. Schmidt. Estimating the Number of Answers with Guarantees for Structured Queries in P2P Databases (Poster). In *CIKM*, 2008. to appear.

[12] M. Karnstedt, K. Sattler, M. Hauswirth, and R. Schmidt. Cost-Aware Processing of Similarity Queries in Structured Overlays. In *6th IEEE International Conference on Peer-to-Peer Computing*, pages 81–89, 2006.

[13] M. Karnstedt, K. Sattler, M. Richtarsky, J. Müller, M. Hauswirth, R. Schmidt, and R. John. UniStore: Querying a DHT-based Universal Storage. In *23th Int. Conference on Data Engineering (ICDE'07), Demonstrations Program*, pages 1503–1504, 2007.

[14] J. Madhavan, S. Cohen, X. L. Dong, A. Halevy, S. Jeffery, D. Ko, and C. Yu. Web-Scale Data Integration: You can afford to Pay as You Go. In *CIDR 2007*, pages 342–350, 2007.

[15] E. Prud'hommeaux and A. Seaborne. SPARQL Query Language for RDF, 2006. W3C Candidate Recommendation.

[16] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, and B. Z. an d J. Kubiatowicz. Pond: The OceanStore prototype. In *USENIX*, pages 1–14, 2003.