Light-weight Internet-scale Universal Storage

[Technical Challenge]

Marcel Karnstedt, Kai-Uwe Sattler
TU Ilmenau, Germany
{marcel.karnstedt|kus}@tu-ilmenau.de

Manfred Hauswirth, Roman Schmidt
EPFL, Switzerland
{manfred.hauswirth|roman.schmidt}@epfl.ch

ABSTRACT

Many new applications, for example Wikis, social networks, and distributed recommender systems, require the efficient integration of decentralized and heterogenous data sources at a large scale. In this paper, we present our vision of a universal storage for RDF-like triple data based on a structured P2P system and the universal relation model as its key enabling technologies to achieve flexibility, robustness, and efficiency for large-scale distributed data storage and query processing. We outline the steps we have already accomplished successfully and discuss a roadmap to achieve the final goal of a practical, light-weight implementation of our storage system.

1. INTRODUCTION

An increasing number of applications on the Web are based on the idea of collecting and combining large public data sets and services. In such *public data management* scenarios, the information, its structure, and its semantics are controlled by a large number of participants and integration and data management functionalities come into existence through the collaborative efforts of the users, i.e., the system's public. Examples of such applications are Wikipedia, social networks, such as friend-of-a-friend networks, or recommender systems.

Despite being distributed or decentralized in respect to data from a conceptual point of view, the supporting infrastructures of these systems still are inherently centralized, as in the original web approach where web servers manage their data locally and only communication and hyper-linking introduce the aspect of decentralization (though the Web itself is decentralized). For example, in Wikipedia articles are edited in a decentralized way, but adding the information permanently to the (central) data collection is done under central control; in social networks, e.g., friend-of-a-friend networks, although inherently decentralized, users typically enter via centralized portals and data management is centralized at the portal. Though often centralization makes sense to maintain full control of the information management, the downsides are bottlenecks and single-point-of-failures, which have to be accounted for by expensive hardware and backup Internet connections.

In this paper, we argue for a decentralization of data management for novel web applications and search engines. This means that information sources are highly distributed, data is described according to heterogeneous schemas, no participant has a global view of all information, and data and service quality can only be guaranteed in a best effort way. Best effort may seem to be a severe limitation at first glance. However, many services we use on a daily basis follow this approach and still provide meaningful service, for example email, DNS, and P2P systems which do not provide any "transactional" service guarantees.

Our argument for decentralization is supported by practical problems of existing infrastructures. For example, Google's long crawling times to collect index data works fine for the current Web with only few highly dynamic and data sources, e.g., news, weather, etc., which are known a-priori and thus can be prioritized. For public data management systems with highly dynamic data this approach, however, is inadequate, as it is unclear which of the many nodes to prioritize. The general problem of data freshness and accuracy remains unsolved, however, and can only be addressed properly if the data is provided and accessed instantaneously from where it is generated, i.e., the participants of the system. This problem can be seen as a modern manifestation of the the old pull vs. push / synchronous vs. asynchronous problem in distributed systems. Additionally, centralized systems as Google require prohibitively high investments into infrastructure, which are another impeding factor.

For such type of public information systems, the P2P approach offers an interesting alternative to existing information system architectures. On the infrastructure side, data is accessed directly at the source, i.e., always fresh, efficient indexing is available, and the systems scale well in terms of nodes and data mounts. Additionally, new systems can be deployed at very low costs as no specialized infrastructures are required as the resources of the participants are being used, high-quality data from the "edge" of the Internet, i.e., the annotated knowledge of the participants, can be made available very easily, and the systems are robust due to their decentralized architecture. On the data level, however, new research problems have to be addressed, the most prominent being: Data may exist in a large number of different schema organizations, it is unclear how trustworthy the data is, and expressivity of queries an possible guarantees (existence, completeness, etc.) are limited at the moment.

Despite these open questions, we argue that global-scale universal distributed storages have a number of important

advantages that outweigh the problems and are a new type of Internet storage system. Our vision is to build a lightweight universal distributed storage for public data/metadata as an enabling backbone technology for storage, which exploits the gigantic storage and processing capacity of the available Internet nodes in the same way as the network layer exploits the worldwide communication devices for routing messages between nodes. In the following sections, we will present the challenges to be addressed and possible solutions.

2. CHALLENGES

In contrast to systems such as OceanStore [14], which aim at secure archival storage for a single data source, we aim at integrating data sources into a universal storage 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 store in the following and classify the challenges along three questions:

(1) How to structure and organize data in massively distributed settings?
(2) How to query data and how to query efficiently?
(3) What is necessary to build a robust and practical solution?

The first question deals with *data organization* and raises two 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. A good choice would be a universal relation model developed in the 1980ies or —as a new incarnation—an RDF-like triple-based model.

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, resolving conflicts should be left to the individual user but be supported by appropriate modeling concepts (e.g., correspondence relationships for schema elements) as well as by explicitly handling schema information as data.

The second question is related to *query processing* with the following challenges:

Expressiveness of queries. Querying data in a large-scale distributed storage requires both classical DB-like queries allowing to restrict and combine data (selection, projection, join, set operations) as well as IR-style queries (e.g., keyword search over all attributes, similarity). In addition, a query language for this domain should support querying schema data (attributes, correspondences) as well and treat this as plain data.

Efficient query operators. Distributed implementations of the query operators should come with worst-case guarantees (e.g., $O(\log n)$ for DHTs) and exploit the features of the underlying infrastructure (e.g., for DHTs hash-based placement, topology-aware routing and multicasting). Furthermore, 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.

Question (3) touches practicability of a large-scale distributed platform. Among others the main challenges are:

Scalability. Certainly, 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.

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, trust, and fairness. 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 important challenges. Secondly, a fair distribution of data and/or load has to be guaranteed in order to avoid negative affect of the overall performance.

In the following sections, we present our approach to meet most of these challenges and discuss ways to address the remaining ones.

3. DATA ORGANIZATION

P2P Infrastructure. Structured P2P overlays offer many features useful for building a universal storage. They scale well, offer logarithmic search complexity in the number of nodes and are based on hashing for data placement, which means that customized versions of existing database algorithms are candidates for fast implementation of a concrete system.

Our DHT of choice is the trie-structured overlay network P-Grid [1]. In P-Grid, nodes are at the leaf level of a virtual binary trie inducing no hierarchy of nodes. The trie is constructed by pair-wise interactions between nodes without central coordination nor global knowledge. While nodes incrementally partition the key space during runtime of the overlay, they keep references to each other to enable prefix-based query routing. A prefix-preserving hash-function assigns data stored in P-Grid to key partitions respectively nodes. While an order-preserving hash function as used in P-Grid keeps semantic relations between data, it requires sophisticated load-balancing to deal with skewed data distributions. P-Grid includes a mature load-balancing technique able to deal with nearly arbitrary data skews [2]. Other DHTs such as Chord use a uniform hash function to achieve load balancing by randomly distributing data in the key space, while P-Grid chooses to load-balancing in favor of more efficient implementation of higher-level query predicates, for example, substring search, efficient range queries [7] and similarity search [9]. While P-Grid can support this through its basic infrastructure, other DHTs require addi-

tional structures, e.g., in Chord an additional trie-structure is constructed on top of its ring-based overlay network to support range queries. Additionally, P-Grid comes with an update functionality with lose consistency guarantees [6]. In contrast to most other DHTs, overlays can easily be merged in a parallel fashion, for example, if two communities discover common interests and want to unify their knowledge or if a network partition had occurred and the DHT fragments were evolving independently for some time. P-Grid is implemented in Java and available from http://www.p-grid.org/.

Triple Storage. In order to face the challenges of data organization, we follow the idea of the universal relation model allowing schema-independent query formulation. However, because exploiting the features of a DHT for fast lookups requires to index all attributes, we store data vertically, similar to the idea of RDF. RDFPeers [4] exploits a similar data organization for RDF data as in our work, but does not address, for instance, similarity-based queries. If we assume relational data, each tuple $(OID, v_1, \ldots, v_n)$ of a given relation schema $R(A_1, \ldots, A_n)$ is stored in the form of n triples

$$(OID, A_1, v_1), \ldots, (OID, A_n, v_n)$$

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 and avoid conflicts. Furthermore, the vertical storage supersedes the explicit representation of null values making the universal relation approach feasible even for heterogeneous data. Obviously, this data storage model is exactly the same layout as RDF – therefore RDF data can be stored seamlessly.

Note that, though we use an OID field, we do not assume unique and homogeneous identifiers for all objects – instead the OID is system generated allowing to group the triples for a logical tuple. Integration tasks, i.e., merging different tuples representing the same real-world object, are expected to be performed on top of this in a user/application-specific way as part of queries.

The hash-based approach of the underlying overlay system allows for inserting each triple multiple times into the DHT using different keys. This is analogous to indexing data in relational systems, as each entry and any combination of the triples' entries (e.g., "hot" attributes) may be chosen, and several kinds of indexes may be implemented (e.g., textual or spatial indexes). This can increase efficiency of query processing by far (see section 4). Moreover, by inserting full triples each time, we introduce a kind of replication on triple level, additionally to replication on peer level, which is essential in DHT-based overlay systems.

By default, we index each triple on the OID, $A_i \# v_i$ (the concatenation of A_i and v_i), and v_i. 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. Like this, efficient reproduction of origin data, as well as access to parts of special interest, is ensured in each situation, as the elements of an origin tuple are stored

(i) clustered – good to achieve low bandwidth consumption and a small number of messages, and
(ii) well distributed – better suited for dynamic situations and load balancing.

By applying according hash functions, in several DHTs similar values are stored at the same peer or neighboring peers, which decreases the efforts incurred in processing range queries, joins, or similarity operations.

Schema Mapping. On top of the data triple storage we additionally allow to store data representing a simple kind of schema mappings in order to overcome schema heterogeneities. In a universal relation model, mappings are simply correspondence links between attributes, whereas different kinds of correspondences can be represented (e.g., "semantically equivalent", "subsumes", ...).

For example, an equivalence mapping between the attributes A_1 and A_2 is represented by a triple

$$(A_1, \mathit{map{:}equiv}, A_2)$$

where *map:equiv* describes the kind of correspondence and A_1 is the identifier of the source attribute. This additional metadata can be queried explicitely by the user – or even automatically by the system to retrieve relevant data without needing the user to interact. Moreover, we think of schema matchers to "crawl" the system (at regular intervals or initiated by the user) and find correspondences (semi-)automatically – thus, the user only needs to join the system, provide his (desired) data and/or schema, and may query remote data at once, without any further intervention and special knowledge! However, such schema matching approaches are beyond the scope of this paper. Instead we refer to the respective work [13].

The introduced data organization helps to deal with some of the before mentioned challenges, because it provides a generic and flexible schema, which can even be extended by meta information to overcome the burden of data heterogeneity. By building the triple store on top of a DHT overlay, we can exploit powerful features of these systems to create a robust, scalable and reliable distributed storage. However, problems like trust and privacy in such environments are treated by the DHT and database research communities, but are far from being solved finally. From the view of data integration, the introduced model provides a wide range of capabilities to utilize integration techniques for dealing with data heterogeneities. But, schemes of different users and communities will still be that heterogeneous that well-known and unsolved problems of this area, e.g., the currently unsatisfyingly developed approaches of automatic schema matching, are still on stage.

4. QUERY PROCESSING

Key lookups, in recent time also range queries on key level and maybe prefix search, are supported by existing overlay systems. In order to support the formulation and processing of DB-like queries we propose a structured query language VQL (Vertical Query Language), which is derived from SPARQL [12], and introduce an according logical algebra. There are several works aiming for the support of structured queries in P2P overlay systems, too. One is PIER [8], where the authors propose relational algebra operators like joins on top of a CAN-based overlay, but no advanced operators like similarity joins or ranking operators. Query operators such as equi-selection, range selection and hash joins, and their implementation, using modified Chord search algorithms are presented in [16].

An example VQL query corresponding to the abstract schema we introduced before looks like:

```
SELECT ?v1,?v3,?v2
WHERE { (?o1,?A1,?v1) (?o2,?A2,?v2)
        (?o1,year,?v3)
        (?A1,map:equiv,?A2)
        FILTER (edist(?A1,title)<2)
        FILTER (edist(?v1,?v2)<3)
      }
ORDER BY ?v3 DESC LIMIT 10
```

In a VQL query, the targeted triples are formulated in braces, where variables are indicated by a question mark.

Optional `FILTER` statements, that may include specialized functions (e.g., `edist(x,y)` calculates the popular edit (Levenshtein) distance between two strings `x` and `y`), are used to provide filter predicates returned triples have to match. Further, the basic construct remembers the structure of SQL queries, including obligatory `SELECT` and `WHERE` blocks, optional statements like `ORDER BY` and `LIMIT`, as well as advanced ones like `SKYLINE OF`.

The semantic of the figured query is not the point, rather we want to present a representative set of the operations the search engine supports: the query represents a string similarity join between the values `v1` and `v2` of two attributes `A1` and `A2`. For `A1`, we only want to select attributes in an edit distance lower than 2 to the term `title`, which stands exemplarily for a filter operation on schema level. The join attribute `A2` is determined using special triples symbolizing schema mappings as introduced in section 3. Finally, we want the data item represented by OID `o1` (this could be, for instance, a car, a movie, etc.) to have an attribute `year` included. The final result is sorted on the values of this attribute, resulting in a *top-N* query due to the included `LIMIT` statement.

The algebra supports traditional "relational" operators $(\pi, \sigma, \bowtie, \dots)$ as well as operators needed to query the distributed triple storage, where the most important are:

- a materialize operation ξ to access triples on leaf level
- another materialize operation ω to "fill" partial tuples with missing attributes and values (implemented as a special join $\bowtie_{OID}$ on the generic object ids)

Operators of both classes can be freely combined and are applicable to schema, instance and metadata level. Furthermore, in order to support large-scaled and heterogeneous data collections, we extend the set of operators by special operators like similarity operators (e.g., similarity join) and ranking operators (e.g., top-N, skyline). Similarity operations are an extremely important and essential part of a universal storage as we propose, and have been considered in, e.g., [3] and [15] before, but both works lack details and an extended experimental evaluation. At the one hand, this applies to data integration tasks on schema level (e.g., attributes representing identical semantic meanings could be named differently), as we briefly discussed in section 2. On the other hand, this equipollently applies to instance level, as such similarity operations are urgently needed for determining schema mappings (e.g., when a new user initially joins the system) and for overcoming problems occurring from erroneous data due to the union of a wide range of individual users and/or communities. But, rather than providing a concrete way of mapping schemes and integrating data, we provide a wide range of operations and techniques to achieve the integration task, which finally is always due to the individual user. Despite this wide functionality, query formulation is based on a simple and small set of VQL clauses.

In our environment, for each logical operator there are several physical implementations available and in development. All of the implemented operators only rely on functionality provided by the overlay system (key lookup, multicasts, provided routing strategies, ...). They differ in the kind of used indexes, applied routing strategy, parallelism, etc. To give an example, in [10] we discussed several possibilities of processing string similarity joins and selections. In this context, we proposed a q-gram index (q-gram: a substring of fixed length q) for the introduced triple storage, which means we additionally insert triples by hashing on the q-grams of an attribute's value and/or name. Based on this, in contrast to "naive" key lookups or range queries, similar strings can be located by requesting a very small set of candidate strings – see, e.g., [9] for details. An enormous advantage of the triple storage is that the processing of each physical operator – analogous to their logical equivalents – is based on the same principles and algorithms on all levels: schema, instance and meta level. Picking up the idea from section 3: automatic schema matchers could utilize provided similarity operators to find correspondences – and so are completely integrated into the physical level of our system as they are into the logical level.

As a representative for the set of physical operators we briefly present our implementation of a skyline operator. Each element of a skyline is not dominated by any other data item with respect to a set of target ranking functions. This operator is very popular in large-scaled data collections, as it is capable of providing a quick but meaningful overview of actual data items. Moreover, the processing of several target ranking functions included in one such skyline query reflects a classical user behavior in distributed storage environments as we propose. On this basis, efficiently implemented skyline operators are a powerful method to attack challenges like scalability and the expressiveness of queries.

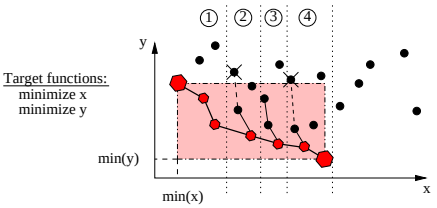

Figure 1: Skyline approach

Figure 1 gives an overview of the implemented skyline processing strategy. Rather than collecting all relevant data at a selected peer that computes the *global* skyline, we ask each peer responsible for a part of the queried dimensions (i.e., attributes) to determine a *local* skyline (in the figure there are 4 peers responsible for distinct partitions of the final skyline, symbolized by a corresponding number in a circle). These local skylines are shipped along with query plans and the last peer involved in processing can compute and return the global skyline. This approach helps to minimize consumed bandwidth and is best suited for parallel processing. Further, we are able to narrow the set of peers potentially responsible for a part of the final skyline: before starting the actual processing of (local) skyline(s), we determine the minimum in at least one preferably "selective" dimension. In the following, we use the values of all other targeted dimensions of the so found data item in order to prune the set of skyline candidates. Figure 1 again illustrate this. If we determine the minimum of dimension x, any point with an y-value higher than the determined one cannot be included in the global skyline (e.g., peer 2 and 4 each can ignore one point of the local skyline if they know about the y-value of $min(x)$). If we determine minima in more than one dimension, we are able to narrow the search space even more – in the figure only the shaded rectangle has to be searched.

The actual processing of skylines is based only on the functionality provided by the underlying DHT overlay. Minima are located using lookups for minimal keys, range queries and prefix searches are utilized to efficiently route skyline queries to involved peers in (quasi-)parallel, and are again based on networking techniques like multicasts.

The physical operators are used to build complex query plans. The processing of these plans can be described as an extension of the concept of *Mutant Query Plans* [11]. For

each physical operator, and thus, for each query plan, we can determine worst-case guarantees (almost all are logarithmic) and predict exact costs [9]. We base these calculations on the characteristics of the used overlay system and the actual data distribution. Combining the costs of single operators we can derive a cost model for choosing concrete query plans. Currently, we are able to enumerate several physical query plans for simple VQL queries, predict corresponding costs, choose the best plan and process it. Coming next we will extend this to arbitrary complex queries and an adaptive processing strategy, repeatedly considering costs and current network situations at each peer involved in a query.

5. IMPLEMENTATION AND EVALUATION

Figure 2 shows the architecture of the implemented system. Based on the P-Grid DHT layer, triple storage functionality is provided by the *TripleManager*, which is called by P-Grid's *StorageService* to store triple data and to process structured queries as describe in the previous sections.

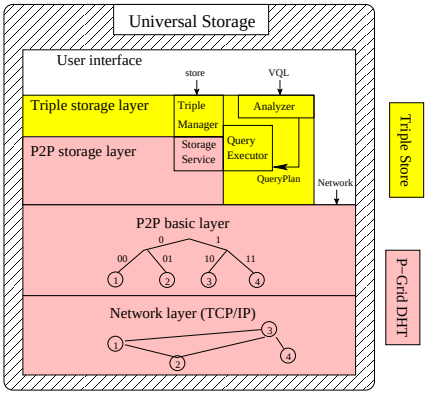

Figure 2: Architecture of application client

A new node joins the universal storage system by starting a copy of the Java client, optionally inserting own data. Distribution of the triples among nodes according to the hash keys, establishment of routing tables, replication and data exchange are provided by the P-Grid layer. Naturally, this requires some time due to the scale of the system and the incurred networking delays. Queries can be issued immediately, but very recent data may only be found after it has been in the system for a certain time (as in any DHT). Queries are formulated in VQL which the *Analyzer* uses to generate logical query plans. *QueryPlans* are wrapped into P-Grid messages and routed to the affected peers in the DHT. The binary key(s) required for this are generated by the plan operator(s). Once a responsible peer receives a query plan, the TripleManager delegates the processing to a local copy of the *QueryExecutor*. Here, logical plan operators are replaced by physical implementations based on cost estimates, and are processed. Partial results are inserted into query plans and shipped along, until a final result is received at a query's initiator.

Currently, we implemented several variations of similarity selections, similarity joins and ranking operators. These can be combined with the introduced *triple-based* operators ω and ξ, as well as simple operators only wrapping P-Grid core functionality, e.g., key lookup. Several new operators, like a Skyline operator, and new variants of existing operators are under development.

We conducted evaluation using the basic system already implemented. In order to be able to evaluate the system we decided to run experiments on PlanetLab [5] to obtain results from large-scale experiments under realistic networking conditions. PlanetLab (http://www.planet-lab.org/) is a global testbed for large-scale experiments with distributed systems. At the moment it consists of approximately 600 nodes geographically distributed over the whole planet running a modified version of Linux to support efficient administration and resource sharing for large-scale experiments. Nodes are connected via a diverse collection of links. When interpreting the results presented in the following, it is important to consider that PlanetLab is shared by a large number of research groups for experiments that are executed in parallel and thus, mutually influence the performance considerably, especially with respect to absolute latency.

The aim of these experiments was to evaluate bandwidth consumption and number of messages as the key performance characteristics in DHT overlays. Furthermore, the experiments give a first proof of correctness for a cost model proposed in [9]. 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 all q-grams and replication each peer was responsible for approximately 900 index entries. The constructed P-Grid tree had a height of 8. We ran 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 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. The following figures show the average number of messages, average bandwidth consumption, respectively, measured per minute at each peer.

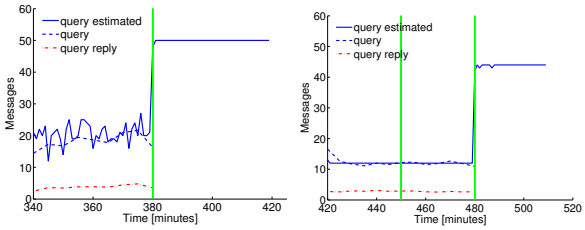

Figure 3: String similarity costs

The left part of 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 a q-gram based join operator and predicted corresponding costs. From time 380-420 the plot of estimated costs also shows the estimations for a more naive join variant based on sequentially querying all peers responsible for an attribute. Due to its inefficient nature, we did not achieve useful results with this operator in the described experimental setup. This is anticipated by the estimated costs as well. As we want to determine the correct relations between costs of different physical implementations, rather than exact costs, these results are fine. The plot shows that we are able to achieve this.

In the right part of figure 3 we present analogous results for similarity selections. Time 420-480 corresponds to q-gram based variants again, whereas for time 480-520 predicted costs for the inefficient sequential implementation are plotted. Similar to the experiments on joins, the plots show

the correctness of our cost model, and that bad performance is predictable a-priori.

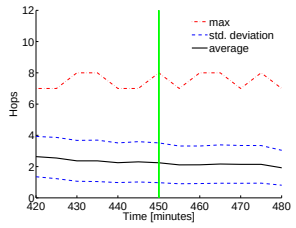

Figure 4: Number of hops

Figure 4 shows the number of hops needed for processing both types of mentioned similarity selections. As the techniques applied here represent basic building blocks for advanced operators, our approach promises to be an efficient implementation of a large-scaled distributed storage, even for skewed data distributions and dynamic and unstable environments.

To the best of our knowledge, a size of 400 peers is more than each system similar to our approach was ever run on. The consumed bandwidth (figure 5) and produced amount of messages for the q-gram based processing are acceptably low. Considering the satisfyingly accurate prediction of costs in an environment as dynamic and unstable as PlanetLab, our approach promises to be a very scalable and performing universal storage solution.

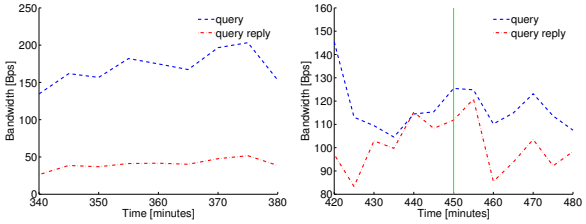

Figure 5: Bandwidth consumption

6. NEXT STEPS

To address the challenges described in section 2 we have defined a roadmap. As a premier area we will conduct further large-scale experiments on PlanetLab. While analytical models and simulations are key planning tools, our experience shows that verification of these models with large numbers of controlled experiments are a key factor to tune the models towards realistic network situations and refine them for more accurate predictions. In addition to these "designed" experiments we use the use cases of the distributed semantic desktop under development in the NEPOMUK integrated EU project (Networked Environment for Personalized, Ontology-based Management of Unified Knowledge, http://nepomuk.semanticdesktop.org/) to construct experiments based on real-world user requirements. The universal storage outlined in this paper is a key part of the distributed infrastructure to be delivered in NEPOMUK.

In section 4 we briefly presented the implemented technique of processing skylines, which was optimized to be efficient in distributed storage systems as ours. But, this operator also is a good example for the open issues in query processing. Interesting challenges in this context are, for instance, how to support the efficient processing of *(i)* a wider range of ranking functions and how to exactly combine them with respect to the DHT (e.g., use *min* at first and advanced rankings afterwards, or vice versa, etc.), and *(ii)* skyline queries over more than two attributes with respect to the traditionally one-dimensional indexes established on top of the DHT. These and similar issues are also acute for other advanced operators, e.g., top-N queries and similarity joins, particularly on schema level.

Another important issue is reliability, which has to be addressed on several levels. On top of the inherent features of a DHT (maintaining redundant links and data replication) we have to ensure also reliable query answers, i.e., guarantee completeness of results. Our current push-based processing strategy is not able to achieve this: if a peer "loses" a query message, portions of the final result could lost. A possible solution is to introduce "heartbeat" messages which are sent to the initiator by each peer participating on the query or a light-weight "transactional" protocol. This would enable the monitoring of the query progress and to estimate the achieved completeness of the result.

We will also investigate better support for data integration. Though attribute mappings and similarity operations, such as similarity join, are helpful for integration purposes, we are interested in (semi-)automatic approaches for deriving and recommending mappings. Because with our data organization scheme both similar attributes (i.e., with similar names) as well as similar values are stored at the same or neighboring peers, a combined schema/instance matching could be performed without crawling the whole network.

7. REFERENCES

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

[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. *SIGCOMM Comp. Comm. Rev.*, 33(3), 2003.

[6] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *ICDCS*, 2003.

[7] A. Datta, M. Hauswirth, R. Schmidt, R. John, and K. Aberer. Range queries in trie-structured overlays. In *IEEE P2P2005*, 2005.

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

[9] M. Karnstedt, K.-U. Sattler, M. Hauswirth, and R. Schmidt. Cost-Aware Processing of Similarity Queries in Structured Overlays. In *IEEE P2P2006*, 2006.

[10] M. Karnstedt, K.-U. Sattler, M. Hauswirth, and R. Schmidt. Similarity Queries on Structured Data in Structured Overlays. In *NetDB'06*, 2006.

[11] V. Papadimos and D. Maier. Mutant Query Plans. *Information and Software Technology*, 44(4), 2002.

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

[13] E. Rahm and P. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4), 2001.

[14] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *USENIX*, 2003.

[15] D. A. Tran. Hierarchical Semantic Overlay Approach to P2P Similarity Search. In *USENIX*, 2005.

[16] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *DBISP2P*, 2004.