Query Processing in a Self-Organized Storage System

Hannes Mühleisen
supervised by Prof. Dr.-Ing. Robert Tolksdorf
Networked Information Systems Group
Freie Universität Berlin
Königin-Luise-Str. 24/26
D-14195 Berlin, Germany
muehleis@inf.fu-berlin.de

ABSTRACT

The amount of scalability and robustness provided by current solutions for the storage and retrieval of data might not be sufficient to support ever-larger web applications. By using swarm intelligence to route operations in a distributed storage system, these limitations can be overcome. However, the possibilities for the efficient evaluation of complex queries in this kind of system are scarce and have not been researched yet. Based on a schema-less data model along with the building blocks for complex queries on this model, I present an approach for complex query processing as part of my PhD work. Here, complex queries are moved through the distributed storage system, while constantly being re-optimized using strictly local information. The approach is described along with an evaluation methodology and a test protocol. My goal is to contribute complex query processing for a fully distributed storage system with unreliable basic read operations and probabilistic directional content routing.

1. INTRODUCTION

A lack of feasible integration mechanisms combined with market forces favoring data silos are currently leading to ever-larger web applications. Many of those applications share the need for a scalable and robust background storage, where the data generated by its users is stored and retrieved from. Single computer systems are having difficulties providing this level of scalability, both to the number of data items as well as to the amount of concurrent requests to the storage layer. Hence, data and/or requests are now commonly distributed onto multiple computers, which are interconnected and serve requests in a co-operative fashion. The degree of distribution and the method used to accomplish distributed operation differ widely, for example, some distributed databases copy all data to "slave" computers, while others partition the stored data between multiple computers and route requests accordingly. In the latter case, various methods can be used to route requests to the nodes storing matching data items. A *federated* approach is using a central directory index server, which keeps track of the storage locations for each data item, and is able to route request (or partial requests) to the nodes storing matching data. However, should this index server fail, the entire system is not able to serve requests any more. More advanced solutions use a fully *distributed* request routing method, in which no node has any special role, and request routing is performed by combining the efforts of several nodes. Here, the loss of nodes does not inhibit the capability of the other nodes to locate a data item and thus serve a request. Examples for this class of systems are the key/value store Dynamo [4] and the column-oriented Hbase [1].

However, previous approaches for distributed storage all faced their limits on skewed distribution of keys within the data and on unfair distribution of queries for this data. These limitations potentially lead to nodes either idling or being overloaded [2]. Our group is in the process of developing a third alternative to the distributed storage and retrieval of data, which is based on swarm intelligence, in particular the foraging behaviour found in several ant species, to provide a data location mechanism in a distributed system. By trading away completeness of the result set for locality transparency, this *swarm-based* system is conceptually able to scale and also adapt to changing data and request characteristics beyond the capabilities of previous approaches [12]. This is achieved by regarding the network of computers to be used to store data as a landscape, and model operations as swarm individuals moving on this landscape. This movement is influenced by virtual pheromones, which are left behind by successful operations.

So far, requests were assumed to only request on data item – regardless of the way data items are structured – at a time. However, data distribution should be transparent to an application making use of the storage system. Stand-alone systems generally provide expressive query languages, for example the Structured Query Language (SQL) for relational data bases, or the SPARQL Query Language for RDF. A similar level of expressivity can also be expected from distributed storage systems. Query processing in federated systems has been described by Bernstein et al. [3] as well as Epstein et al. [6], where query processing can rely on a central instance with universal knowledge. For fully distributed systems based on global organization structures, the efficient evaluation of complex queries represents a more recent problem, but several approaches already exist [8, 7, 14, 16].

For our swarm-based class of distributed storage systems, several unique properties provoke the design of a new method for complex query evaluation. In particular, the request routing method used in the swarm-based system is not able to explicitly pin-point the location of any data item; instead the request is directed to a neighboring node within the storage network possibly either storing the data item or forwarding the request further in the direction of the node actually storing the data item. Also, the employed swarm algorithm of ant foraging makes use of a small random error factor in those forwarding decisions for various reasons. While beneficial for most cases, this random factor can lead to failed requests. A method

for query processing in this class of system must be able to reflect these two impairments and is expected to still be able to process a complex query efficiently enough.

From these prerequisites, I am able to formulate a research question for the following work: Can complex queries be evaluated efficiently in a swarm-based distributed storage system? I will define and discuss my approach on the following pages. In particular, Section 2 defines the distributed storage environment, restrictions on the used data model, and complex queries. Section 3 describes the proposed evaluation methodology aimed to show the feasibility and performance of my query processing approach. Finally, Section 4 concludes this paper and describes the next steps in the course of this research.

2. SELF-ORGANIZED DISTRIBUTED QUERY PROCESSING

As our group has shown in previous research [12], the method used by several ant species use to forage for food has been found suitable to implement a scalable distributed storage system. Here, ants on the search for food start off from their nest on a random walk on a landscape. They continue on this random walk until they encounter food. A portion of the found food is then picked up and carried back to the nest, which is located using its distinct nest scent. On the way back to its nest ants leave a trail of chemical pheromones behind, which form a pheromone path from the food source to the nest. Now, if other ants on their random walk in search for food encounter this pheromone trail, they are inclined – but not forced – to follow it in the direction of the food as determined by comparison with the direction of the nest scent. Should they also find food following this path, they will further intensify the pheromone intensity, increasing the inclination of other ants to also follow this path. As pheromone trails also evaporate over time, old trails for example leading to depleted food sources are discarded soon.

The ant foraging method represents a positive enforcement mechanism which has been shown to be able converge approximating hard problems such as the Travelling Salesman problem [15, 5]. Ant colonies have been shown to be able to scale to billions of individuals, a property which depends on the basic principles of simplicity, locality and dynamism: Ant decisions are not too complicated, their adherence to the defined rules does not require complex reasoning. Also, all decisions can be taken based on strictly local knowledge and limited visibility. Finally, as they are only inclined, not forced, to follow paths, the colony is able to find new food sources.

We have developed a distributed storage system based on this approach. In our system, a number of computers ("nodes") interlinked by network connections are regarded as a landscape. Each node is connected to a limited set of other nodes, and stores a portion of the data to stored in the entire system. Each node offers an interface for client applications, able to store and retrieve data items. Storage and retrieval requests are modeled to be a swarm individual able to autonomously move about on that landscape, typically requiring multiple hops over network connections between nodes in order to find a data item. On each node, requests compute the node to be visited next from the nodes directly connected to the current node using the pheromone trails present. Virtual pheromone trails are laid tracing back the path successful operations, aiding current and future operations in their routing decisions. A small random factor is included in the routing decisions, modeling the ant's inclination to follow pheromone paths. As a replacement for a nest scent, operations carry a path history of visited nodes. If requests are successful, this path is traced back and the virtual pheromones are intensified, yielding the described positive enforcement for each stored data item. Pheromone values are distinct for each data item, since finding each data item represents its own optimization problem from every node. However, pheromone values are aggregated by the system for each neighbor connection to limit the amount of memory occupied by them. Pheromone values are numeric representations of the data item, for example generated by a hash function.

Using this swarm-inspired approach, the storage network can autonomously optimize request routing for all stored data items, given a sufficient number of subsequent requests for a data item [11, 12]. One main benefit is the complete in-transparency of the actual location of stored data inside this system. This allows the system to adapt very quickly to changing data characteristics and fluctuations in data popularity, which is often impossible in other approaches [2]. For example, should one node be overloaded with requests while neighboring nodes stand idle, the data items stored on this node may readily be redistributed using an arbitrary heuristic between neighboring nodes. This movement does not require the involvement of different parts of the storage network at all, allowing the concept to scale. Also, should nodes exceed their storage capacity due to massive amounts of data, excess data can also be moved off to neighboring nodes or completely different areas of the storage network. Again, nodes not directly involved do not need to be notified and all such data organization decisions can be taken on each storage node independently, making this storage system truly self-organized, potentially outperforming other distributed approaches.

Two main restrictions for queries to this class of system became apparent: First, the system is never able to determine the location of a data item before an retrieval operation has reached the node storing the data item. Second, through the random factor involved in routing decisions, retrieval operations may not be successful every time. Nonetheless, since these restrictions are a key to achieving the desired scalability and adaptability, other requirements such as complex query support have to be designed accordingly. Also, evaluations showed the success rate of the system to be very high in practice, thus making this storage system potentially interesting to applications [12]. However, so far only exact-match single key retrieval operations are possible, clearly falling short of the expressivity expected from a full-featured distributed storage system. To define queries more expressive than simple matches for unstructured data items based on a single key, we have to define the data model further.

2.1 Data and Query Model

For the purpose of this research, we assume a tuple-based data model. A tuple is regarded as an ordered list of basic data types such as text strings or numbers. Formally, a tuple is defined as $tuple := (t_1, t_2, \ldots, t_n)$, where individual entries can be accessed using their index $t_i := \{e_j \in t | j = i\}$. The following tuples are an example of data expressed within this model:

```
(BMW,isA,CarManufacturer)
(BMW,residesIn,Munich)
(Daimler,isA,CarManufacturer)
(Daimler,residesIn,Stuttgart)
(Munich,isA,City)
(Munich,hasInhabitants,1330000)
(Stuttgart,isA,City)
(Stuttgart,hasInhabitants,601000)
```

A well-known special case of this data model is the Resource Description Format (RDF) [10], where all data is expressed as triples. This model has the advantage to be adaptable both to key-value storage or a relational model. Regardless of the actual language

used to formulate queries, we define a set of the two basic building blocks for complex queries, selection and join operations. Both selection and join operations depend on a basic read operation accessing the physical storage layer. This basic read operation is only able to match a single tuple entry against a predefined value, since the routing decision has to be given a single key to determine the neighbor node where the operation should be forwarded to. The selection operation mends this restriction and is able to handle a so-called tuple pattern, where an arbitrary number of values can be pre-defined, and all undefined values are treated as wild cards. Both the basic read as well as the selection operation return a set of tuples as their result. The following list details the operations:

- $read_{t_i,i}$ – This basic read operation will lead to the creation of a virtual individual moving through the storage network returning tuples matching the given value t_i on the tuple index i. For example, the tuples `(Stuttgart,isA,City)` and `(Stuttgart,hasInhabitants,601000)` could be searched for by calling $read_{Stuttgart,1}$ on the above data set. Very importantly, due to the properties of the swarm-based system, this read operation is susceptible to retrieval failures and may have to be restarted. Failure is modeled as a probability p_{fail}, which denotes the probability that the read operation will fail.

- $select_{pattern}$ – The selection operation supports the retrieval of tuples using more than one fixed value. It is given a pattern, which can be regarded as a template for matching tuples. For each entry of tuples matched against the pattern, the pattern either defines a value or specifies a wild card, defined as $pattern := (p_1, p_2, \ldots, p_n), p_i \in (*, t_i)$. For example, $select_{*,isA,City}$ would be a pattern matching a single tuple `(Stuttgart,isA,City)` from the above example. The selection can be implemented by intersecting the result sets of read operations for each pattern entry specifying a value.

- $join_{ts_1,ts_2,j_1,j_2}$ – The join operation is used to combine two tuple sets ts_1 and ts_2 by comparing the values for each tuple using the join indices j_1 and j_2. For each pair of tuples with equal values for the join indices from the two tuple sets a new result tuple is created by appending the second tuple without the matching entry to the first tuple. For example, joining the tuples `(a,b,c)` and `(c,d,e)` with $j_1 = 3$ and $j_2 = 1$ would produce `(a,b,c,d,e)`.

From these building blocks, a wide range of complex queries can now be constructed. It should be noted that projection (removal of tuple entries not needed further) is not included in the list of operations for simplicity reasons. Also, values may always be projected at a later stage. Obviously, *no* schema definition is required in this data model, even though schema information could help with the querying process. However, schema information would represent a global state which would have to be distributed reliably between all nodes to be of use. This would be very expensive in terms of the number of transmissions required and thus be hindering scalability.

An example for a fairly complex query on the example data set is the query printed in Fig. 1 using a tree representation. The query is retrieving the number of inhabitants of the city a car manufacturer resides in.

2.2 Proposed Solution

To describe the evaluation of these complex queries in the context of our swarm-based distributed storage system, I have chosen the reference architecture for distributed query processing defined by Kossmann [9]. I will continue with discussing the need to adapt components within this architecture to the environment described here. This widely adapted reference architecture consists of the following components:

1. *Parser* – The query is translated from a syntactical representation into an internal data structure according to the query language specification. This component does not need to be adapted.

2. *Query Rewrite* – Query optimizations regardless of the physical state of the system, for example replacing complex language constructs with sets of smaller operations. This component also does not require adaption for the described environment.

3. *Query Optimizer* – Possible query execution plans are enumerated and costs for their evaluation are calculated using a cost model. From all plans and their estimated costs, the best plan is selected for execution. In the described environment, this is no longer possible, as the information required to construct a reliable cost model is not present on any single point within the system. For example, the cardinality of result sets of read operations with specific input are unknown and could only be determined by actually executing this operation, which is not feasible. Hence, a more dynamic approach to query optimization is necessary.

4. *Plan* – The plan is a precise specification as to how the query is to be evaluated. For example, the plan specifies the exact order of basic read operations followed by join operations. The plan is represented in an internal data structure which is also required in the described environment, leaving this component unchanged.

5. *Plan Refinement / Code Generation* – The plan selected for processing the query is translated into an executable plan, which is often a distinct computer program. Since we are not able to determine the optimal plan due to a lack of information, code generation is not possible beforehand, rendering this component unnecessary.

6. *Query Execution Engine* – The execution engine provides implementations for the basic operations needed to support the query language. This is also required within the described environment, requiring no conceptual changes in this component.

7. *Catalog* – The catalog is designed to pin-point any data item to a specific node in the distributed storage system. In the case of a relational data base, the catalog would be able to provide the query processor with the node storing a particular relation (or parts of it). This is not possible in the described environment due to the second restriction, only allowing requests being routed towards the node actually storing requested data. Hence, this component is also subject to change.

From analyzing each component involved in query processing, the query optimizer as well as the catalog were identified to be in a need for adaption in this scenario. The query optimizer lacks the information needed to optimize queries beforehand, and a catalog cannot contain information about the data stored in the entire network due to scalability reasons. Furthermore, single retrieval operations as part of a complex query may fail and would have to be restarted. From the previous work on the subject of distributed complex query processing, two approaches seem fit for this scenario:

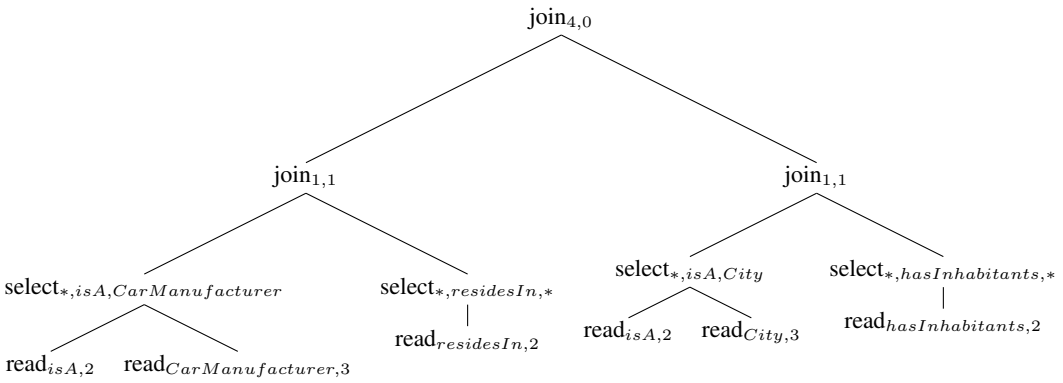

Figure 1: Example Query – Tree Representation

Battré described complex query processing in an distributed RDF store built atop a distributed hash table, which moved the query processing between nodes if seemed fit according to the expected communication costs [2]. Papadimos et al. propsed the concept of *mutant query plans* for the use in distributed XML data bases [13], where the query plan is constantly optimized as new data and thus more information to base optimization on becomes available.

In my concept, query plans are both moved from node to node as well as constantly optimized. Query plans start moving through the network in their entirety towards the node where the data to be read first is stored after limited initial optimization (comparable to Step 2 – Query Rewrite). On its way, parts of the plan are resolved, evaluated, and re-optimized if needed constructing the cost model from strictly local information and heuristics, then finally yielding both an optimal plan as well as a complete result. Hence, the steps 3 through 7 are constantly repeated as more information relevant to optimization becomes available. To continue the example from the complex query pictured in Fig. 1, the order in which the 6 different $read$ operations should be carried out is completely unknown at first. Query evaluation starts using an execution order calculated using the limited information available locally. The query then starts to move through the storage network on the search for tuples fitting the read operation selected for evaluation. However, the execution order is constantly re-evaluated using the present pheromone paths stored on each node as the basis for an ad-hoc cost model. In general, the execution order is changed if the algorithm has sufficient information to assume the evaluation of the query would be improved with a probability larger than a specific threshold, e. g. 0.5, e.g. using a heuristic based on the present pheromones paths.

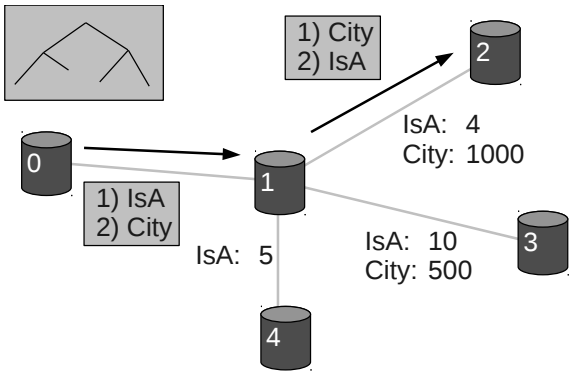

Figure 2: Reordering example

Fig. 2 shows a single step in the processing of the example query from Fig. 1: Multiple nodes 0 through 4 are connected, each connections has virtual pheromones assigned to them. The connection leading from node 1 to node 2 has two virtual pheromones, *isA* with intensity 4, and *City* with intensity 1000. Pheromones are also assigned to the other network connections. The read requests for triples with *isA* and *City* are part of the basic read operations for the example query, which has been sent to node 0 for evaluation. Lacking more information, node 0 decides that the read operation for *isA* should be evaluated before the read operation for *City*. The query now is forwarded to node 1, where the continuous query optimization takes place. Based on the present pheromones, the inverse execution order is considered to be closer to the optimal evaluation order. Hence, the query processing is continued using *City* as the first read operation, and based on the present pheromones likely forwarded to node 2.

Mutable and moving query plans are a solution for efficient query processing in the presence of only directional routing. To handle the possible failures of read operations, a second technique is required: Operations trace the nodes they have visited during retrieval operations to support pheromone laying. These path traces can be used to recover from failed retrieval operations as well. Routing failures become apparent whenever the operation becomes stuck in on a single node or starts following circular routing pattern without finding new results for any scheduled read operation. These conditions can be detected using local algorithms from the path traces, and the query evaluation operation can be resumed, for example from start or – more sensibly – from the last node to deliver results.

If tuples are found for one of the read operations, two alternatives are possible: First, matching tuples could replace the read operation in the query execution operator tree and sent along with the query. This would enable instant joining of the tuples carried along with tuples found on other hosts. However, large partial result sets would seriously impede the speed of the query operation due to increased transmission times. The second alternative would be to only store a pointer to the matching tuples and execute the entire query once every operator has been resolved to a set of nodes. While this enables further optimizations, since the storage system is able to move tuples in order to optimize their storage locations, tuples found on a node might not be there later.

After repeating the described process over several hops inside the storage network, the query evaluation is expected to converge on a near-to-optimal execution order, especially due to the determination of the possible benefit of a re-ordering. Should this not be the case, further analysis of the encountered problems can be used either to improve my approach.

3. EVALUATION METHOD

The evaluation of distributed algorithms bears several challenges, for example, changing delays in transmission between nodes can have an impact on the performance of a distributed algorithm. For the presented method of distributed query processing, two main evaluation criteria are obvious: First, the method is required to produce correct and usable results to applications running queries. Second, the scalability of the proposed method has to be shown. Here, scalability is dependent on the number of nodes that participate in the processing of a single query. The approach can only be considered scalable, if this number is considerably below the amount of nodes in the storage network. Hence, the evaluation plan is set to incorporate both criteria, hopefully showing both the correct function as well as the scalability of the proposed method. Due to the various statistical properties inherent in the approach, a formal proof is impossible. A first evaluation will thus be based on a statistical analysis. Since the method can be fully implemented, further evaluation can be performed through applications of the method inside a network of different computers, closely approximating the environment in which my approach is expected to be deployed to. Here, the test protocol is designed as follows:

1. Construction of a storage network with a set of nodes, with each node being connected to a non-zero number of neighbor nodes.
2. Storage of a pre-defined data set of tuples inside the storage network; each node will be responsible for storing a subset of the data set.
3. Repeated evaluation of a pre-defined set of queries on the storage network. The result set and the number of nodes participating in the generation of the result sets are recorded. Since the correct results for the queries can be calculated beforehand, the result set will give an insight on the correctness of the algorithm.

To determine the influence of the storage network structure and size, several different classes of network structures for different numbers of nodes will be constructed, and the evaluation repeated for each storage network. For example, a star-shaped storage network might be better suited for my approach than a randomly connected network. Also, the number of neighbors for each node might play an important role. In order to determine the performance of my approach for increasingly complex queries, the evaluation will also be repeated for both a synthetic and real-world data sets with corresponding queries. The synthetic data set and queries will allow the evaluation of the query processing for several constructed cases, while the real-world data and query set will provide some insight on the performance of my approach in a setting as realistic as possible.

While the result set quality can be determined by comparing result sets with the correct result determined beforehand, the amount amount of nodes participating in the generation of a result is evaluated differently: For each query, the evaluation is set to determine the optimal query plan by generating a global view over the storage network normally not present. From this optimal solution, I will be able to assess the overhead generated by my approach due to its intentional lack of global information. For a comparison with the state of the art, I will also record the amount of nodes participating in resolving the test queries using a state-of-the art algorithm for example using a distributed hash table as described in [2].

4. CONCLUSION

From the amount of data already present or expected in the future the need for distributed storage and retrieval of information was motivated. The different paradigms in the design and operation of distributed storage systems were described, introducing our concept of a self-organized distributed storage system based on swarm intelligence. Within this class of system, the possibilities for the efficient evaluation of complex queries are scarce and have not been researched yet. The two main restrictions from the system's design are location intransparency and failure-prone basic retrieval operations. A basic schema-less data model along with the building blocks for complex queries, selections and join operations were introduced. Based on a widely accepted reference architecture for distributed query processing, I have identified the areas of required work and presented an adaption of the concept of mutable query plans. This concept is based on local optimizations of a query execution in order to reach an efficient query execution. My evaluation methodology of statistical analysis and a test protocol for the proposed method over different storage network structures as well as data and query sets was described. The main contribution of my research would be the described approach of executing complex queries in a fully distributed storage system with unreliable basic read operations and probabilistic directional content routing.

4.1 Next Steps

The next steps in my PhD research are to formulate a detailed algorithm for complex query evaluation within the data model and system environment I have described in this paper. Based on this algorithm, I will perform a statistical analysis on the likelihood that this algorithm will produce a near-optimal query execution in typical cases. Based on this, a prototypical implementation will be used to execute various runs of the described test protocol, including a large-scale evaluation using real-world data.

5. ACKNOWLEDGMENTS

I would like to thank the anonymous reviewers for their insightful and detailed comments. This research has been supported by the "DigiPolis" project funded by the German Federal Ministry of Education and Research (BMBF) under the grant number 03WKP07B.

6. REFERENCES

[1] Balakrishnan, Kaashoek, Karger, Morris, and Stoica. Looking up data in P2P systems. *CACM: Communications of the ACM*, 46, 2003.

[2] D. Battré. *Efficient Query Processing in DHT-based RDF Stores*. PhD thesis, Technische Universität Berlin; Fakultät IV - Elektrotechnik und Informatik. Institut für Telekommunikationssysteme, 2008.

[3] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, Dec. 1981.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (21st SOSP'07)*, pages 205–220, Stevenson, Washington, USA, Oct. 2007. ACM SIGOPS.

[5] M. Dorigo and T. Stützle. *Ant Colony Optimization*. The MIT Press, Cambridge, Massachusetts, 2004.

[6] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. In *19 ACM*

SIGMOD Conf. on the Management of Data, Austin TX, May 1978. Also published in/as: UCB, Elec.Res.Lab, TR.No.ERL-M78/18, 1978.

[7] Harren, Hellerstein, Huebsch, Loo, Shenker, and Stoica. Complex queries in DHT-based peer-to-peer networks. In *International Workshop on Peer-to-Peer Systems (IPTPS), LNCS*, volume 1, 2002.

[8] M. Karnstedt, K. Hose, and K.-U. Sattler. Distributed query processing in P2P systems with incomplete schema information. In Z. Bellahsene and P. McBrien, editors, *DiWeb*, pages 34–45, 2004.

[9] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv*, 32(4):422–469, 2000.

[10] F. Manola and E. Miller. RDF primer. World Wide Web Consortium, Recommendation REC-rdf-primer-20040210, Feb. 2004.

[11] R. Menezes and R. Tolksdorf. A new approach to scalable linda-systems based on swarms. In *Proceedings of ACM SAC 2003*, pages 375–379, 2003.

[12] H. Mühleisen, A. Augustin, T. Walther, M. Harasic, K. Teymourian, and R. Tolksdorf. A self-organized semantic storage service. In *Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS2010), preprint available at http://digipolis.ag-nbi.de/preprint/iiwas2010-s4-preprint.pdf*, 2010.

[13] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *CIDR*, 2003.

[14] P. Rösch, K.-U. Sattler, C. von der Weth, and E. Buchmann. Best effort query processing in DHT-based P2P systems. In *ICDE Workshops*, page 1186, 2005.

[15] T. Stützle and M. Dorigo. A short convergence proof for a class of ant colony optimization algorithms. *IEEE-EC*, 6:358–365, Aug. 2002.

[16] C. Treijtel. AmbientDB: Complex query processing for P2P networks. In M. H. Scholl and T. Grust, editors, *VLDB PhD Workshop*, volume 76 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.