Best Effort Query Processing in DHT-based P2P Systems

Philipp Rösch[1] Kai-Uwe Sattler[1] Christian von der Weth[1,3] Erik Buchmann[2]
[1]Dept. of Computer Science and Automation, TU Ilmenau, P.O. Box 100565, D-98684 Ilmenau, Germany
[2]Dept. of Computer Science, University of Magdeburg, P.O. Box 4120, D-39106 Magdeburg, Germany
[3]Current address: University of Karlsruhe, Am Fasanengarten 5, D-76131 Karlsruhe, Germany

Abstract

Structured P2P systems in the form of distributed hash tables (DHT) are a promising approach for building massively distributed data management platforms. However, for many applications the supported key lookup queries are not sufficient. Instead, techniques for managing and querying (relational) structured data are required. In this paper, we argue that in order to cope with the dynamics in large-scale P2P systems such query techniques should be work in a best effort manner. We describe such operations (namely grouping/aggregation, similarity and nearest neighbor search) and discuss appropriate query evaluation strategies.

1. Introduction

The existence of millions of nodes connected to the Internet at the one hand and the need to manage huge public datasets in emerging applications in a fair manner at the other hand raises the question, if and how can we exploit the idea of structured Peer-to-Peer (P2P) systems for managing structured data. Such P2P systems promise a high scalability and robustness due to avoiding centralized components and the absence of single point of failures. Applications which can benefit from these features are among others: managing and indexing public datasets in e-Sciences, e.g. genome data or data from astronomy, and managing and querying metadata for the Semantic Web or more precisely Web Service descriptions as a basis for a distributed UDDI. In this paper, we argue that in order to cope with the dynamics in large structured P2P systems appropriate query techniques should work in a *best effort* manner.

Structured P2P systems in the form of distributed hash tables (DHT) are a promising approach for building data management platforms for such scenarios. However, for more advanced applications such as sketched above a simple keyword-based lookup operation is often not sufficient. Thus, the main challenges are first to manage structured data (e.g. relational data) using a DHT and second to process queries consisting of operations such as join, range selections and grouping / aggregation. These challenges were recently addressed, e.g. in [11, 20] by distributing tuples in a DHT and implement relational algebra operators exploiting the hashing scheme. However, the approaches lack the comprehensive consideration of three important problems:

- Because of the large scale of the system and the number of the contributors, we cannot assume that data will be clean and unambiguous. For example, equivalent entries may be spelled differently from different authors. Therefore, we have to deal with inconsistencies at instance level requiring appropriate *best effort operators*, e.g. similarity selection, nearest neighbor search, top-N queries and joins.
- The decentralized and highly dynamic nature of P2P systems makes it nearly impossible to apply classic query planning techniques, where an optimal plan is chosen in advance by taking cost estimations based on statistics into account. Instead we need *best effort query processing strategies* allowing to adapt to the current (load) situation and which are robust to network changes.
- Because of the P2P nature, there are no guarantees about the availability of data or service. Thus, exact or complete results cannot be guaranteed as well.

By "best effort" we mean that we do not aim for exact results or guarantees but instead try to find the best possible solution wrt. available (local) knowledge.

In this paper, we present results of our work on addressing these issues. Starting with a brief introduction in our query processor for a DHT we discuss the implementation of grouping/aggregation, similarity selection and nearest neighbor search operators as well as two alternative query evaluation strategies based on a stateless approach as well as on the idea of eddies.

The remainder of this paper is structured as follows: Section 2 reviews related work. Section 3 introduces our data fragmentation scheme. Based on the query operators described in Section 4, Section 5 discusses distributed pro-

cessing strategies in DHT. Section 6 provides an evaluation by the means of experiments, and Section 7 concludes with an outlook to future work.

2. Related Work

Distributed hashtables (DHT) are able to cope with very high numbers of parallel transactions that process huge sets of (key,value)-pairs. DHT follow the peer-to-peer paradigm, i.e., they consist of many autonomous nodes, there is no central coordinator and global knowledge is not available. Prominent DHT appliations are distributed search engines, directories for the semantic web or genome data management (cf. [7] for an exhaustive list).

Examples of DHT are Content-Addressable Networks (CAN) [14], Chord [18], Pastry [15] or P-Grid [1]. The proposals mainly differ in the topic of the key space and the contact selection, i.e., how to distribute the (key,value)-pairs among the peers and which are the nodes a peer communicates with.

The key spaces of Chord [18] or CAN [14] are organized as circles in one (Chord) or multiple (CAN) dimensions, each peer is responsible for a certain part of the key space (zone), and the nodes communicate with peers responsible for adjacent zones. P-Grid [1] is based on a virtual distributed search tree. Each node is addressed with a subtree-ID. For each level of the tree, each node maintains a reference to another peer in the same subtree, but whose ID branches to a different subtree in the deeper levels. [8] features a detailed survey of the various approaches. Our DHT-based query processor is applicable to any DHT. But key space topologies different from CAN require an adoption of the data fragmentation scheme presented in Section 3.

DHTs are originally designed to perform operations of a hashtable, e.g., `put(key, value)` or `value = get(key)`. But soon the database community has started to adapt techniques from distributed and parallel database systems [5, 10] to (rather simple) DHTs.

An approach with a stronger relation to modern DHT systems is the PIER project. [11] presents a system architecture based on CAN, and discusses several distributed join strategies.

A detailed discussion of query processing strategies for Chord-based P2P systems is given in [20]. Here, the authors propose a framework of operators and their implementation by using modified Chord search algorithms. Another P2P-based query processing approach is AmbientDB [4]. It exploits Chord-based DHTs as clustered indexes for supporting selection operators, but integrates non-P2P components as well.

Because of the hashtable-like nature of a DHT, exact-match selections can simply be implemented on top of the existing DHT lookup operation. However, many applications depend on a more sophisticated query algebra. In order to address the problem of range operations, [2] proposes space filling curves and DHT flooding strategies for query processing. Similar work is described in [9] based on locality sensitive hashing.

3. CAN & Data Fragmentation

A variant of distributed hash tables are Content-Addressable Networks (CAN [14]). Each CAN node is responsible for a certain part of the key space, its *zone*. This means, the node stores all (key, value)-pairs whose keys fall into its zone. The key space is a torus of Cartesian coordinates in multiple dimensions, and is independent from the underlying physical network topology. In other words, a CAN is a virtual overlay network on top of a large physical network. The keys of the CAN are represented as lists of d coordinates $k = \{c_1, c_2, \cdots, c_d\}$. The CAN uses a hash function to map the keys of the applications to its own key space, e.g., $f(''Bach'') = \{0.3425, 0.94673\}$. We refer to the hash function in conjunction with the assignment of zones as *fragmentation scheme*. In addition to its (key, value)-pairs, a CAN node also knows all peers which are responsible for adjacent parts of the key space. A query in CAN is simply a key lookup in the key space, its result is the corresponding value. A node answers a query if the key is in its zone. Otherwise, it forwards the query to another node by using *Greedy Forwarding*: Given a query that it cannot answer, a peer chooses the target from its neighbors so that the Euclidean distance of the query key to the zone of the peer in question is minimal, i.e., the designated peer is closer to the key than the current one.

Figure 1 shows an example key space of a CAN, together with a query. The key space of P_i is shaded in grey. In addition to its key space, P_i also knows the nodes N_1, N_2, N_3 and N_4. Assume P_i has issued a query for $k = (0.2, 0.2)$, i.e., it wants to retrieve the corresponding value. Since P_i is not responsible for this key, it uses Greedy Forwarding and sends the query to N_3. N_3 in turn is not responsible either, thus the procedure recurs until P_r is reached which returns the query result to P_i.

Storing relational structured data in a DHT raises the questions of how to partition the relations, which attributes should be used for applying the hash functions and can we benefit from a clustering of certain relations. A viable solution has to be found somewhere between the extreme cases:

1. distribute tuples only according the relation ID, i.e., each relation is completely stored at exactly one peer or
2. distribute tuples according their key value such that tuples are partitioned among all peers. In addition, in or-

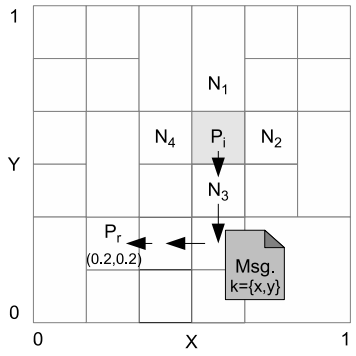

Figure 1. Content-Addressable Networks

der to support range and nearest neighbor queries efficiently, tuples of the same relation should be stored at neighboring peers.

We have evaluated different strategies and found the *reverse bit interleaving* approach very promising. The main idea is to use two hash functions h_R and h_K for the relation ID $\mathcal{R}$ as well as the key value t_K of the tuple t. Now the hash values are bit-wise concatenated: $h(\mathcal{R}, t_K) = h_R(\mathcal{R}) \circ h_K(t_K)$. Finally, the resulting bit string is split into d coordinates by using Bit 0, d, $2d$, ... for the first coordinate, Bit 1, $d+1$, $2d+1$, ... for the second and so forth. Thus, tuples are stored along a Z curve in the CAN (Figure 2). Obviously, the hash functions have to be order-preserving in order to identify peers for a given domain by using the Z value, i.e., subsequent tuples (e.g., 0,1,2; "tea","tee") have to be managed by the same peer or an immediate neighbor in the key space. Furthermore, this scheme contains several adjustable parameters. For example, by introducing a "spread factor" the length of the Z curve interval and in this way the number of peers storing one relation can be adjusted. [16] presents an outright description of reverse bit interleaving.

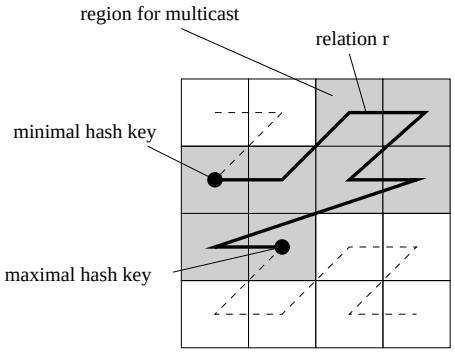

Figure 2. Z curve in CAN

Figure 2 sketches such a Z curve in a CAN. The curve covers the zones containing fragments of the domain of the primary key values. Suppose a node issues a range query. Now the Z curve enables the peers to ask only the nodes (shaded in grey in the figure) which could have parts of the query result. In detail, the query issuer transforms the range expression to a minimal and a maximal hash key on the Z curve, and uses a multicast to send subqueries to all nodes between the minimal and maximal hash keys.

Note, that this storing scheme can be used not only for the tuples but also for indexing. In this case, only a pointer (relation ID, key value) is stored together with the key at the peer. This pointer can be used to retrieve the actual item from another location.

4. Query Operators

Using a data fragmentation scheme as described in Section 3, query operators can work in parallel, i.e. intra-operator or partitioned parallelism can be achieved. In [16] we have presented how classical plan operators such as selection, join and grouping can be implemented in this way. The main idea is to exploit the DHT for routing purposes. Therefore, at the base level we have added two primitives to the DHT API:

- `send_message`(z,m) sends a message m (e.g. a query plan) to the peer responsible for the zone containing point z on the Z curve,
- `send_multicast`($z_{\min}$,$z_{\max}$,m) sends the message m to all peers maintaining zones in the Z curve interval $\langle z_{\min}, z_{\max} \rangle$.

Based on these primitives we have implemented the following query operators:

- exact match selection on the key attribute (simply a hash table lookup),
- range selections on the key attribute (send a multicast message to peers of the Z curve interval determined by hashing the selection ranges),
- other selections (full table scans, implemented using multicast),
- symmetric hash join (re-inserting tuples of both input relations in a temporary join relation),
- "ship where needed" join (where tuples of one relation are sent to the peers storing possible join candidates and which are identified by applying the hash function to the join attributes).

All these operators are exact operators. In the following we will focus on best effort approaches for further operators.

Grouping. A rather simple form of a best effort solution is used for grouping / aggregation. In order to reduce or even avoid query state information during processing we use a stateless processing strategy (see the following section), where a plan "travels" through the network and the operators are subsequently replaced by their results. However, for blocking operations such as grouping / aggregation we do not know when the input relation is completely processed and when we can compute the aggregate(s).

Assume a grouping operator $\gamma_{F,G}$ with the aggregate function F and the grouping attributes G which is applied on a relation or intermediate result sets of another operation. Then the grouping is processed as follows:

1. Each node performing $\gamma_{F,G}$ sends its tuples to a peer determined by applying the hash function to the value of G.
2. Each "grouping" peer collects incoming tuples with the same value of G and applies F. In order to deal with the problem of unknown end of input we follow the idea of early aggregation, i.e. after a certain time or a number of processed tuples each grouping peer computes the aggregate and sends the intermediate result back to the query initiator.
3. The query initiator now can output and/or refine the resulting aggregates and stop the processing at a given time.

Similarity selection. Another import class of best effort operations are similarity operators such as similarity selection. One approach to support this operation in a hash-based storage system is to use q-grams for implementing edit distance-based similarity search. Here, string-valued keys are split into distinct q-length substrings (usually with $q = 3$ or $q = 4$). These substrings are inserted into the DHT together with the tuple itself or the tuple key as a reference to the tuple. Thus, a similarity selection on the attribute A using the search key s is evaluated as follows:

1. Construct a set of q-grams from the search string s. Note, that not necessarily all q-grams have to be considered. Instead, one can choose only a subset of non-overlapping q-grams, i.e. for a threshold e only $e + 1$ q-grams are required [12], where q-grams with a high selectivity are chosen for preselection [17].
2. For each q-sample s_i send a selection request to the peer responsible for the zone containing the point $h(\mathcal{R}, s_i)$. This is part of the basic functionality of the DHT.
3. Each receiving peer whose q-gram key matches the search key s sends a message with the corresponding tuple t to a "collector" peer identified by $h(\mathcal{R}, t_K)$.
4. The collector peers receive the candidate strings matching the q-grams and perform the final filtering.
5. If a given similarity threshold is reached the tuple is returned as a result of the similarity search.

Another possible approach is to sent all q-grams and apply a count filtering technique as described in [6].

Nearest Neighbor Search. In order to be able to process NN queries, the keys of the tuples stored in the DHT have to be generated with a hash function that preserves the neighborhood between similar keys of the application. For instance, a distributed seach engine would use a hash function that maps "Bethoven" to a key with a small distance to the key of "Beethoven". A proper hash function for keyword search would generate the key $k = \{f(0), f(1), \cdots, f(d-1)\}$ by suming up the ASCII values of the characters $c = \{c_0, c_1, \cdots, c_l\}$ of the keyword c for each dimension d of the key space:

$$f(n) = \sum_{i=0}^{l/d} c_{(i \cdot d+n)} \cdot v$$

Here, the extent of v is a tuning parameter that adjusts the distance of adjacent keywords.

If a peer P_i wants to obtain a set of s tuples closest to a query key k, it issues a NN query $\sigma_{NN}(k, s)$. Now our protocol works as follows:

1. Forward the query to the node responsible for the query key. This is part of the basic functionality of the underlying CAN protocol (cf. Section 3).
2. Return the set of s tuples closest to k to the issuer of the query. Since the key space is partitioned, the correct NN result set may come from multiple peers. In order to allow to refine the result, a set of neighbors, whose zones are closer to k than the distance between k and the most distant tuple in the result, is returned as well. For example, $P0$ in Figure 3 is responsible for k. Given $s = 6$, $P0$ returns the tuples in the circle and all of its neighbors except $P4$.
3. Estimate the precision of the returned data set by dividing the volume of the part of the data space that is already queried by the volume of a field with center k and radius $dist(k, k_s)$. In Figure 3, this is the grey surface area divided by the area of the circle. Stop if the precision satisfies the demand.
4. Refine the result set by asking a node obtained in Step 2, which was not queried before and whose zone has the largest overlapping with the relevant field. Continue with Step 2. In Figure 3, the second peer queried would be $P6$.

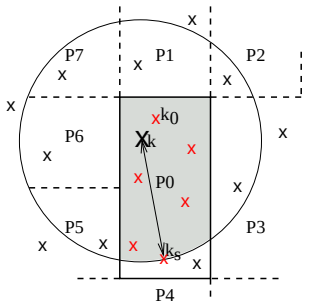

Figure 3. Fragmented NN result set

5. Processing Strategies

Strategies for processing complex queries in DHTs have to consider the inherent characteristics of these systems, i.e. only local knowledge about neighbors as well as unforeseen appearance and disappearance of individual peers. That means, that the query planner cannot rely on the existence of statistics or assume that all peers will respond to a query. Thus, best effort processing strategies are required.

A first approach to deal with this situation is to use a stateless processing approach similar to the idea of mutant query plans [13]. By the term "stateless" we mean that beside the plan no explicit information about the execution state of a query is needed and no peer has to wait for the input from another peer. This works as follows:

1. A left-deep query tree is constructed. This plan is sent to all peers which potentially provide data required by the leftmost operator of the tree. These peers can be identified by applying the hash function to the selection condition of this operator.
2. Each of these peers processes the plan in postorder and executes all operators which are local processable.
3. If an operator cannot be processed locally (e.g. a join), the remaining plan and the intermediate result are sent to the next peers which are again identified by applying the hash functions to the intermediate tuples.
4. If the root of the plan tree was processed the results are sent back to the query initiator.

Using this strategy we can cope with the dynamics in the network: Because the processing peer for the next non-local operator is determined at runtime, changing or not-responding peers do not interrupt the processing. However, there is no guarantee that the result set will be complete. Thus, an important task for future work will be to estimate the completeness of results.

Though, this approach provides robustness it has a major drawback: the query plan is built in advance and makes assumptions about cardinalities and costs in order to choose a certain join order. In large federated and shared-nothing databases, like in P2P environments, typical parameters of classic query optimization are not available or are subject to widely fluctuations. So, traditional static query optimization and execution techniques can lead to less effective results in such environments.

Therefore, we have investigated a more adaptive approach by following the idea of eddies [3]. An eddy is a pipeline for routing tuples to operators of the plan based on runtime statistics such as queue length, selectivity or load. In our implementation an instance of an eddy can run at each peer which participates on a given query. Because we want to avoid any kind of global information and coordination, each tuple (or packet of related tuples) has to keep two things: First, a "ToDo" list containing the operators of the plan which can currently executed on the associated tuple(s). As a result, the appearance is independent from of the original query plan. Second, a "ready" bit for each operator. This bit indicates if the operator is already processable in order to stick to the transformation rules of the relational algebra. Both the ToDo list and the tuple(s) are packed in a message package. This technique allows a more flexible selection of executing peers as well as distributed operators, which is very appropriate for P2P environments.

Tuple routing in a distributed system addresses two issues:

- *operator routing* (Which operator should be executed next?)
- *peer routing* (Which peer should process the tuple?).

For operator routing the follow strategies can be chosen:

- *random* (the next operator is chosen randomly),
- *priority* (to each operator a priority value is assigned which allows heuristics such as "selection first"),
- *input queue length* (each operator contains an input queue; the length of this queue is a measure for processing costs),
- *selectivity* (as proposed in [19] the selectivity of an operator is learned using a ticket approach), and
- *next-join* (if tuples have to be reinserted e.g. for performing a join, one can choose the join first for which the distance of the home peer is minimal).

In all cases, the decision is made only locally because queue length or tickets are known only for the local processed operators. This means, that different peers processing the same query can come to different decisions.

Peer routing is a result of the distributed operators. It allows to migrate the execution of operators to other peers, especially for load balancing purposes. However, for some

operators (e.g. joins) the processing peer is already determined by the operator routing. Thus, only for unary operators (selection, projection, sorting etc.) peer routing is possible. Here, the following strategies can be applied:

- *random choice*,
- *round robin*,
- *best connection*, i.e. the tuple is sent to the neighbor with the fastest connection, and
- *lowest load*, i.e. the tuple is sent to the neighbor with the lowest working load.

Figure 4 illustrates the steps of the eddy-based query processing in a DHT. Note, that in order to reduce the communication costs, we try to group tuples with the same destination peer and apply peer routing only if the local load exceeds a certain threshold.

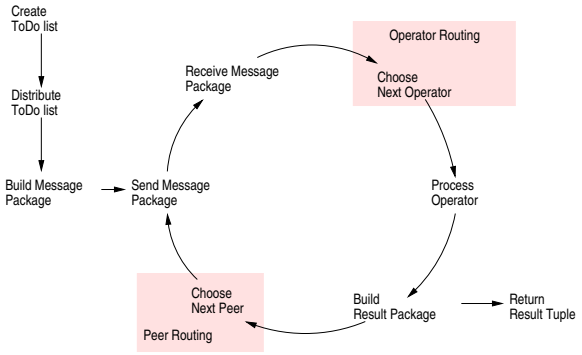

Figure 4. P2P eddy at work

6. Evaluation

We have evaluated the presented operators and strategies in several experiments. The main questions we tried to answer were

1. Are large DHTs suitable for managing and querying structured data?
2. Does our approach fulfill the expectations with regard to scalability?
3. How can the best effort operators and strategies in a P2P system compete with exact approaches based on global knowledge?

The experiments were performed using our query processing engine implemented in Java on top of a CAN prototype. We used network sizes ranging from 1,000 up to 10,000 peers, the TPC-H dataset as well as simulated load. Because of the lack of space we can give here only selected results.

In the first experiment we have investigated the fragmentation scheme by comparing the Z-curve fragmentation with different spread factors to the simple “relation per peer” strategy using a mix of join queries. The results for a selected spread factor in Figure 5 show, that even though the number of required hops of the Z curve approach is around 10 times higher than for the relation/peer strategy, the processing time (measured using a simulation clock) is only 10% of the time. Furthermore, the processing time grows only marginally with an increasing number of peers.

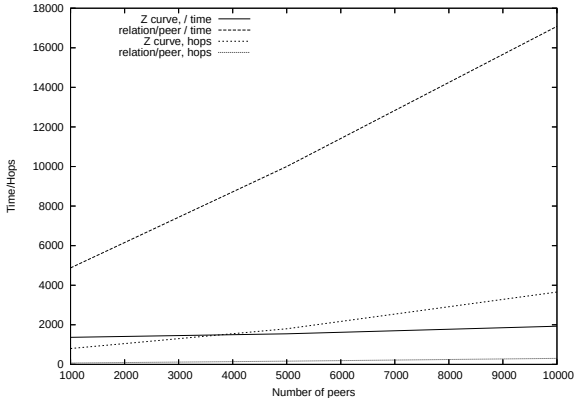

Figure 5. Fragmentation and Scalability

We have evaluated the NN protocol by experiments in a 4-dimensional CAN consisting of 10,000 peers. Here we ran one million queries on an uniformly distributed dataset of 10 million keywords. Figure 6 presents the effort to obtain a result set of a certain quality: the graphs show the relation between the number of peers queried and the precision of the result for different sizes of the result set. It is shown that – except for very large result sets – more than 80% of the query results come from 2 peers, i.e. the expenses for processing NN queries are moderate in settings such as ours.

For the eddy approach we have investigated the impact of the different routing strategies as well as scalability and a comparison with a centralized approach. As shown in Figure 7 by increasing the number of peers the processing time stays nearly constant whereas the number of hops grows. However, if we look at the ratio between hops and number of peers, we can observe that the overall network load decreases.

In a further experiment we compared the different strategies of the operator routing (Figure 8). As expected the random selection of the next operator induces to the worst results, according to time as well as to hops. On the contrary *priority*, *input queue length* and *selectivity* shows equal results. This effect is caused by distribution of the operators.

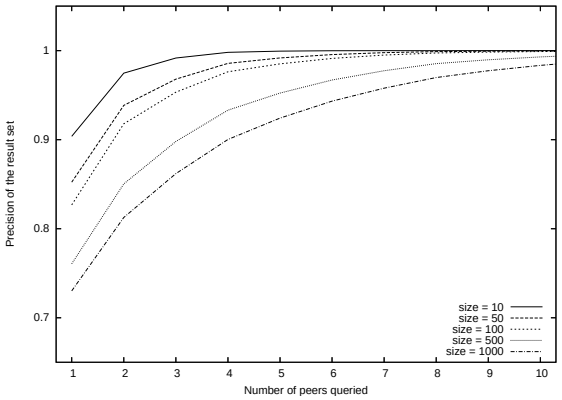

Figure 6. NN Queries: Precision vs. Effort

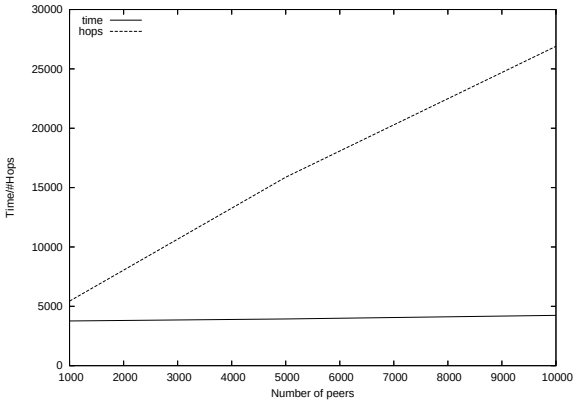

Figure 7. Scalability of the Eddy Approach

The parameter *input queue length* and *selectivity* are just known for the local instance of an operator on each peer. In order to come to an explicit decision for the next operator based on the *input queue length* or *selectivity*, these parameters must be known for all operators of the current ToDo list. This means, each operator of the list has to be processed once at least on the current peer. Otherwise, the eddy uses the *priority* as fallback. Especially in large environments this fallback occurs in most cases.

Finally, we have compared the distributed P2P eddy to two centralized solutions collecting all statistics (Figure 9). We wanted to show that a centralized eddy is not adequate for those systems.

In the first implementation of a centralized eddy, the central peer appears just as coordinator. In the first step, all tuples are sent back to this peer. The execution of the different operators, especially joins, is performed by other peers. Nevertheless, both measures (time and hops) increase. This is caused by two reasons: First, the local load of the coordinator peer is high. Second, all message packages must be forwarded to the initiator peer, although most of the packages do not contain any parts of further result tuples. Both local load and hops affect the processing time.

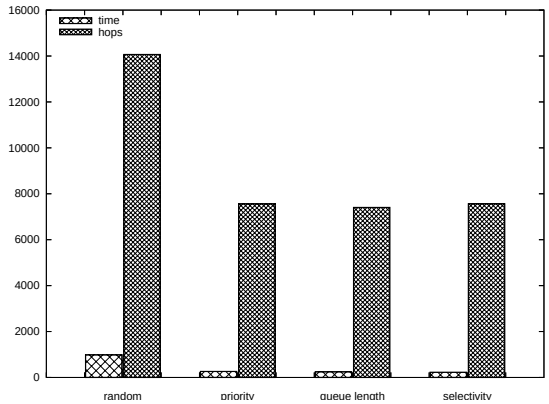

Figure 8. Operator routing strategies by comparison

In a second solution of a centralized eddy, the designated "eddy" peer is the coordinator as well as the executor of the operators. So this strategy corresponds to simple data shipping. Due to the significant increase of the load of the coordinator, the processing time grows in the same way. In order to reduce the number of hops, we can use a simple optimization: All the messages can be sent directly to the coordinator peer, because it is known to all other peers in this solution. A routing through the CAN is not necessary.

Considering the processing time, the P2P eddy outperforms the centralized solutions significantly. In addition, in a large-scale system a central eddy becomes a bottleneck.

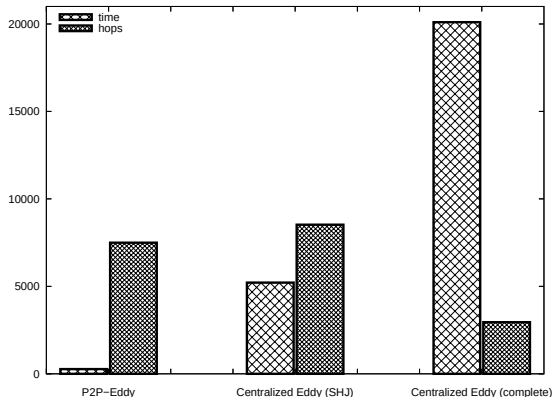

Figure 9. P2P Eddy vs. Centralized Eddy

7. Conclusion and Outlook

Best effort query processing is an important challenge on the way to an SQL-like query processor on top of structured P2P networks. Because of the absence of global knowledge, data inconsistencies and the massively distributed P2P architecture, best effort query processing is a very complex task.

In this article we have presented a set of best effort query operators and processing strategies that promise to cope with these challenges. We have evaluated our approach by the means of extensive experiments. Though these results are very promising, more work is required that addresses further issues of best effort operators. This includes similarity joins based on our presented similarity operations, top-N queries as well as approximate range queries. In addition, we want to investigate in different load scenarios and fair load balancing.

References

[1] K. Aberer. P-Grid: A Self-organizing Access Structure for P2P Information Systems. In *Proc. of CoopIS 2001, Trento, Italy*, pages 179–194, 2001.

[2] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Systems. In *Proc. of the 2nd Int. Conf. on Peer-to-Peer*, 2002.

[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. ACM SIGMOD 2000, Dallas, Texas*, pages 261–272, 2000.

[4] P. Boncz and C. Treijtel. AmbientDB: Relational Query Processing over P2P Network. In *Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2003), Berlin, Germany*, 2003.

[5] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 35(6):85–98, 1992.

[6] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *Proc. VLDB 2001, Roma, Italy*, pages 491–500, 2001.

[7] S. D. Gribble et al. The Ninja Architecture for Robust Internet-scale Systems and Services. *Computer Networks (Amsterdam, Netherlands: 1999)*, 35(4), Mar. 2001.

[8] K. Gummadi and et al. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of the SIGCOMM 2003*. ACM Press, 2003.

[9] A. Gupta, D. Agrawal, and A. Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Proc. of the 2003 CIDR Conference*, 2003.

[10] A. Hameurlain and F. Morvan. An Overview of Parallel Query Optimization in Relational Databases. In *Proc. of Int. Workshop on Database and Expert Systems Applications (DEXA 2000)*, pages 629–634, 2000.

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

[12] G. Navarro and R. Baeza-Yates. A Practical q-gram Index for Text Retrieval Allowing Errors. *CLEI Electonic Journal*, 1(2), 1998.

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

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *ACM SIGCOMM 2001*, pages 161–172, 2001.

[15] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[16] K. Sattler, P. Rösch, E. Buchmann, and K. Böhm. A Physical Query Algebra for DHT-based P2P Systems. In *Proc. 6th Workshop on Distributed Data and Structures (WDAS'2004), Lausanne*, 2004.

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

[18] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001 Conference*, 2001.

[19] F. Tian and D. J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *Proc. VLDB 2003, Berlin, Germany*, pages 333–344, 2003.

[20] P. Triantafillou and T. Pitoura. Towards a Unifying Framework for Complex Query Processing over Structured Peer-to-Peer Data Networks. In *Int. Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2003), Berlin, Germany*, 2003.