Querying the Internet with PIER

Write-up by: Laura Tolosi
Supervisor: Prof. Dr. Gerhard Weikum

1. Introduction

Achieving scalability is one of the goals of the database research community at present. The Internet is estimated to have a few hundreds of nodes, yet the largest database systems in the world scale up to at most a few hundred nodes. Supporting large databases is still a challenge because of the lack in the degree of distribution. The main goal is for databases to scale over Internet, thus making easy for applications (e.g. e-commerce) to develop.

Scalable databases mean scalable data size, speed, workload and transaction cost. Three primary factors for operational databases are: huge number of concurrent users, the need for continuous availability and extremely large stored data volume.

We define in what follows some requirements needed for (linear) scalability:

Size-up. This means that if a database size increases by a factor of X, given a constant hardware configuration, then the query response time should increase by no more than a factor of X.

Speed-up. If the hardware configuration's capacity is increased by a factor of X, then the query response time should decrease by no less than a factor of X.

Scale-up. If the workload on the system is increased by a factor of X then the system should be able to keep the query response time unmodified by increasing the hardware capacity by no more than a factor of X.

Transaction cost. There are two considerations with transaction cost in a scalable system. First, workload increases should not increase the transaction cost. Second, if data size increases by a factor of X, transaction cost should increase by no more than a factor of X.

Since traditional database systems fail (for the moment) to achieve scalability, the authors claim to have found a compromise solution to this problem: PIER (Peer-to Peer Information Exchange and Retrieval) – a query engine that comfortably scales up to thousands of participating nodes. The key idea is to marry traditional relational database query processing with peer-to-peer inspired technology (the engine is built on top of a Distributed Hash Table).

2. Querying the Internet

Applications

A motivating application for massively distributed database functionality is presented in what follows.

Despite of the many "questionably" legal Peer-to-Peer systems, there exist natural (and legal) applications for *in situ* distributed querying, where data is generated in a standard way in many locations instead of being centralized. Warehousing can be unattractive for many reasons. They are best suited for historical analysis, while some applications prefer live data,

they are expensive to administer, requiring large storage capacity and large bandwidth to scale.

An interesting application is *Network Monitoring*. Network protocols like IP, SMTP, HTTP tend to have standard representations, thus providing with structured data that could be used for query processing. To be more precise, network behaviors can be characterized by "fingerprints": sequences of port accesses (to detect port scanners), port numbers and packet contents (for buffer overrun attacks or web robots), or application-level information on content (for e-mail spam). If standard servers and applications would distribute structured "fingerprints" over the network, then intrusion detection could be possible by simply querying for frequent fingerprints that match recently experienced attacks. One can determine the threat level in the network, detect similar fingerprints, how many reports on attacks exist, etc.

For example, say that a fingerprint table is distributed among users. A summary of widespread attacks can be obtained by computing an aggregation function over this table:

```
SELECT I.fingerprint, count (*) AS cnt
FROM intrusions I
GROUP BY I.fingerprint
HAVING cnt > 10;
```

For refined results, assume that some users are more reliable in the network than others, thus their reports being given more credit. An additional reputation table is thus distributed among users. The results will be weighed according to the reputations of the publishers:

```
SELECT I.fingerprint, count (*) * sum (R.weight) AS wcnt
FROM intrusions I, reputation R
WHERE R.address = I.address
GROUP BY I.fingerprint
HAVING wcnt > 10;
```

Design Principles

Traditional databases fail to achieve massive distribution because of a series of assumptions, such as consistency, availability, etc. PIER uses relaxed design principles in order to achieve scalability:

Relaxed consistency

Since the goal is to design a query engine for Internet-scale database systems, transactional consistency is given up in favor of availability and tolerance of network partitions (it should work on whatever subset of the network is reachable). As compensation, PIER provides best-effort results and measure them using precision, recall, etc.

Organic Scaling

PIER tries to avoid an architecture where *a priori* allocation of a data center is required. This is both expensive and an obstacle against scalability. Peer-to-Peer –like technology is a good means to achieve organic scaling.

Natural habitats of data

Data is no longer loaded into a database, instead it remains in its "natural habitat" – e.g. a file system. Wrappers must provide with structured data to be used by the query engine. This structured information is temporarily copied in the query system's storage space.

Standard Schemas via Grassroots Software

Internet-scale distributed database systems assume availability of structured information among thousands of users. At a first glance, this is not a very easy task to accomplish and can appear discouraging. One interesting idea of the authors is to use the structured information naturally produced by popular, widely used software.

3. Content Addressable Network (CAN)

PIER is build on top of a distributed hash table. The authors chose a particular architecture for a DHT, namely Content Addressable Network. The design and functionality of CAN have been introduced by Silvia Rtnasamy, Paul Francis, Mark Handley, Richard Karp and Scott Shenker from University of California, Berkeley [2]. They argue that CAN is scalable, fault-tolerant and completely self-organizing, thus appearing to be a useful instrument for PIER. Nevertheless, PIER has been deployed also over another DHT, Chord.

The basic operations performed on a CAN are insertion, lookup and deletion of (key, value) pairs. CAN is composed by many individual nodes, each storing a *zone* of the entire hash table. In addition, a node holds information about a small number of "adjacent" zones in the table. Requests (insert, lookup or delete) for a particular key are routed by intermediate CAN nodes towards the CAN node whose node contains that key. CAN is completely distributed (it requires no centralized control, coordination or configuration), scalable (nodes maintain only a small amount of control state that is independent of the number of nodes in the system), and fault-tolerant (nodes can route around failures).

The design centers around a virtual *d*-dimensional Cartesian coordinate space. This coordinate space is completely logical and is not related to any physical coordinate system.

At any point in time, the entire coordinate space is dynamically partitioned among all the nodes in the system such that every node owns its individual, distinct zone. For example, Figure 1 shows a 2-dimensional coordinate space partitioned among 5 CAN nodes.

The virtual coordinate space is used to store (key, value) pairs as follows: to store a pair (*k, v*), key *k* is deterministically mapped onto a point onto a point *P* in the coordinate space using a uniform hash function:

$$f(k) = (f_1(k), f_2(k), \ldots, f_d(k))$$

The corresponding key-value pair is then stored at the node that owns the zone in which point *P* lies.

To retrieve an entry corresponding to a key *k*, any node can apply the same deterministic function to map *k* onto a point *P* and then retrieve the corresponding value from the point *P*. If this point is not owned by the requesting node or by one of its immediate neighbors, the request must be routed through the CAN infrastructure until it reaches the node in whose zone *P* lies.

A node learns and maintains as its set of neighbors the IP address of those nodes that hold coordinate zones adjoining its own zone. This set of immediate neighbors serves as a coordinate routing table that enable routing between arbitrary points in the coordinate space.

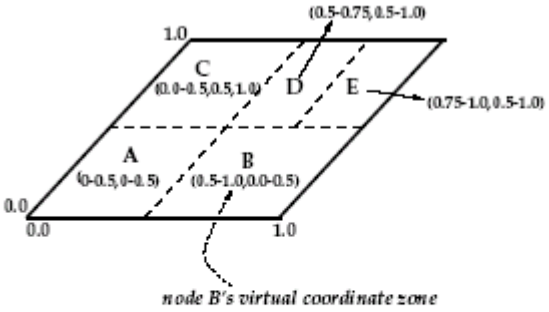

Figure 1: *Example 2-d coordinate overlay with 5 nodes*

Routing in CAN

Intuitively, routing in a Content Addressable Network works by following a straight line path through the Cartesian space from source to destination coordinates.

A CAN node maintains a coordinate routing table that holds the IP address and virtual coordinate zone of each of its neighbors in the coordinate space. In a *d*-dimensional space, two nodes are neighbors if their coordinate spans overlap along *d-1* dimensions. This logical neighbor state is sufficient to route between two arbitrary nodes in the coordinate space. A CAN message includes the destination coordinates. Using its neighbor coordinate set, a node routes a message towards its destination by simply greedy forwarding to the neighbor with coordinates closest to the destination coordinates. Figure 2 shows a sample routing path.

For a *d*-dimensional coordinate space partitioned into *n* zones, the average routing path length is thus:

$$\frac{d}{4} n^{\frac{1}{d}}$$

and individual nodes maintain *2d* neighbors. These scaling results mean that for a *d*-dimensional space, the number of nodes can grow without increasing per node state while the path length grows as:

$$O(n^{\frac{1}{d}})$$

Since many different paths exist between two points in the space, even if one or more of a node's neighbors were to crash, a node would automatically route along the next best available path. Further mechanisms of repairing the routing scheme in case all neighbors in a certain direction of a node fail at a moment are described in [2].

Joining in CAN

To allow the CAN to grow incrementally, a new node that joins the system must be allocated its own portion of the coordinate space. This is done by an existing node splitting its zone in half, retaining half and handing the other half to the new node. This process takes three steps:

- First the new node must find a node already in the CAN

A list of existing nodes in the CAN is supposed to be published and available.

- Next, using the CAN routing mechanisms, it must find a node whose zone will be split.

The new node randomly chooses a point *P* in the space and sends a join request destined for node *P*. This message is sent into the CAN via an existing CAN node. The message is routed until it reaches the node in whose zone *P* lies. This current occupant then splits its zone in half and assigns one half to the new node. The split is done by assuming a certain ordering of the dimensions in deciding along which dimension a zone is to be split, so that zones can be re-merged when nodes leave. The key-value pairs from the half-zone to be handed over are also transferred to the new node.

- Finally, the neighbors of the split zone must be notified so that routing can include the new node.

Having obtained its zone, the new node learns the IP addresses of its neighbors from the previous occupant. Similarly, the previous occupant updates its routing table. Finally, their neighbors must be informed on this reallocation of the space. Figures 2 and 3 show an example of node 7 joining the CAN.

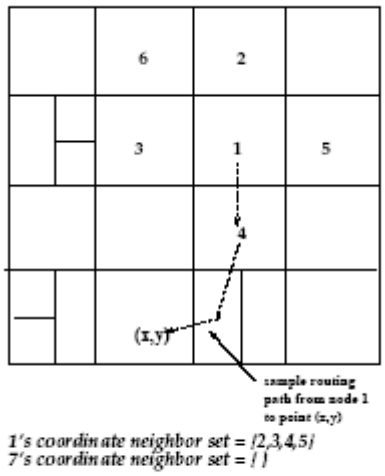

Figure 2: *Example 2-d space before node 7 joins*

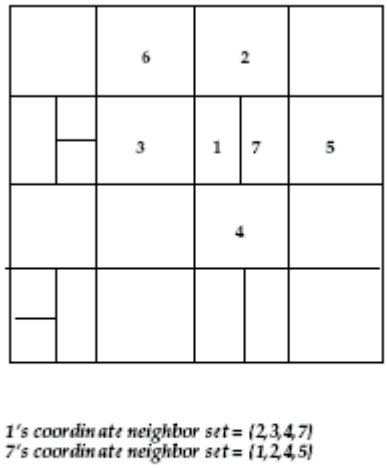

Figure 3: *Example 2-d space after node 7 joins*

Node departure, Recovery and CAN maintenance

When nodes leave CAN, one need to ensure that the zones they occupied are taken over by the remaining nodes. The normal procedure for this is for a node to explicitly hand over its zone and the associated (key, value) database to one of its neighbors.

If the zone of one of the neighbors can be merged with the departing node's zone to produce a valid single zone, then this is done. If not, then the zone is handed to the neighbor whose zone is currently the smallest, and that node will temporarily handle both zones.

The CAN needs to be robust to node or network failures. This is handled through an immediate takeover algorithm that ensures one of the failed node's neighbors takes over the zone. However, in this case the (key, value) pairs will be lost until the state is refreshed by the holders of the data.

Refined algorithms are proposed to solve the problem of high fragmentation of the space (see [2]).

4. PIER Architecture

In what follows, PIER's architecture will be presented. PIER is a three-tier database-style query-engine, as shown in Figure 4. Applications interact with the PIER Query Processor, built on top of a DHT. An instance of each DHT and PIER component is run on each node.

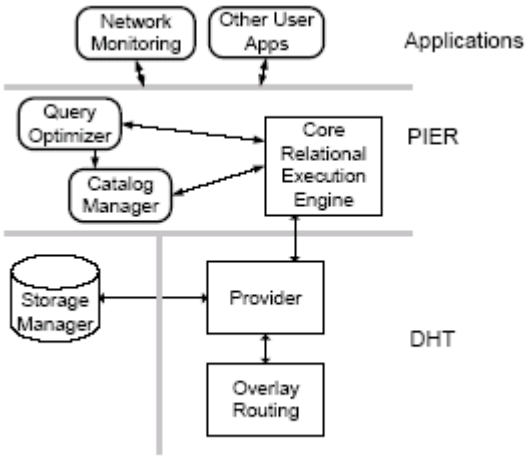

Figure 4: ***PIER architecture***

As some applications have already been presented in section 2, next sections will present the other two layers in detail.

4.1.DHT design

For the simulations presented in the article, the authors used CAN as a DHT design, with the dimension of the space being $d = 4$. However, the reason for which CAN is chosen is not well argued. Although one of the main intended qualities of CAN's design is scalability (as stated by its authors, in [2]), the designers of PIER argue that the scalability results could be improved via another DHT.

The DHT design is factored in three layers: Routing Layer, Storage Manager and the Provider.

Routing Layer

The routing layer is responsible for mapping keys into the IP address of the node currently responsible for that key. The API contains 3 functions.

lookup function has a key as input parameter and returns (asynchronously) the IP address of the node in the system responsible in whose zone the key is hashed.

join function is called when a new node wants to join the DHT and it knows a landmark already in the network.

The *locationMapChange* callback notifies higher levels asynchronously when the set of keys mapped locally has changed, frequent situation as nodes join and leave the network.

lookup(key) → ipaddr
join(landmark)
leave()
locationMapChange()

Table 1: *Routing Layer API*

Storage manager

This layer is responsible for the storage of the hashed data while the node is connected to the network.

The data in the DHT is highly distributed among users, thus in many applications each machine will store a relatively small amount of data. Communication with the storage manager is realized via a simple API

store(key, item)
retrieve(key) → item
remove(key)

Table 2: *Manager API*

Provider

The provider ties the routing layer and the storage manager together. It also provides with an interface to applications.

Naming scheme for DHT-based data: each object has a *namespace*, *resourceID* and an *instanceID*. The DHT key of an object is the hashing of the *namespace* and the *resourceID*.

*Namespace*s semantically correspond to relations (tables), for query processing.

*ResourceID*s are typically related to the primary key of the relation to which it belongs.

Namespaces need not be a priori defined; they are created in the moment the first tuple is inserted and destroyed when all tuples expire. Since a *namespace* and a *resourceID* do not necessarily uniquely define a tuple, a random *instanceID* is generated by the application in order to make a difference.

The semantics of *put* and *get* functions in the provider API are straightforward. One observation is maybe needed: the get function may return multiple items, as it is key-based, not instance-based.

The "lifetime" argument of the function *put* gives the DHT a bound on how long an item should be stored after receipt. Producers of the data can prologue the life of one piece of information by invoking *renew* function. This instrument is also useful when nodes fail and information is temporarily lost, in order to restore it.

When a query is run, the *multicast* function provides the means to contact all users that hold data in a particular namespace.

Data stored locally can be scanned by the *lscan* iterator. When run in parallel on all nodes serving a particular namespace, this serves the purpose of scanning a relation.

The *newData* callback informs the application when a new item belonging to a particular namespace has arrived.

get(namespace, resourceID) → item
put(namespace, resourceID, instanceID, item, lifetime)
renew(namespace, resourceID, instanceID, item, lifetime) → bool
multicast(namespace, resourceID, item)
lscan(namespace) → iterator
newData(namespace) → item

Table 3: ***Provider API***

4.2.Query Processor

As the goal of PIER is to achieve scalability, the query processor supports simultaneous execution of multiple operations, pipelined together.

The query processor is 'read-only'; it implements operators for selection, projection, distributed joins, grouping and aggregation. Insert, update, delete are performed directly via the DHT interface.

In order to hide network latency, the query execution is done as much as possible in parallel. Operators are not linked together like in traditional databases. Instead, operators produce results (push) that are immediately enqueued for the next operator (pull).

As a future improvement, the authors admit that a query optimizer is necessary and could be added on top of the existing query processor. Further extension could provide insert, update and delete of items and tables facilities directly in the query processor.

As stated previously, scalability is achievable, but on the expense of transactional consistency.

The *reachable snapshot* is defined as the set of data published by reachable nodes at the time the query is sent from the client node.

Instead of that, the relaxed notion of correct data set is defined by the *dilated reachable snapshot*: union of local snapshots of data published by reachable nodes, where each local snapshot is from the time of query message arrival at that node.

The correct data set is now slightly time-dilated, which may lead to a low consistency, due to failures and partitions.

5. DHT – Based Distributed Joins

The join algorithms used by PIER are adaptations of regular parallel distributed join algorithms.

The equi-join algorithm is essentially a DHT-based version of the pipelining *symmetric hash join*. Consider, for the sake of the example that a join on tables *R* and *S* is issued by a query initiator and that these tables are stored in the DHT under separate namespaces: N_R and N_S.

The query initiator multicasts the query to all nodes that hold items in the *R* or *S* namespaces. All these nodes scan their relations (with *lscan*) to find tuples that mach the query. Each tuple that satisfies the local selection predicates is copied (with only the relevant columns remaining) and rehashed in a new (temporary) namespace, say N_Q.

When rehashing, the *put* function is used:

put (*namespace*, *resourceID*, *instanceID, item, lifetime*)

where:

namespace = N_Q
resourceID = the join attributes concatenated
item = the actual record, with additional tag regarding the namespace it comes from (*R* or *S*)
lifetime = the amount of time the record will be available in the *Q* namespace.

Once an item is produced, it is immediately *pushed* into the temporary namespace (which plays the role of a queue). The beautiful part is that the hashing scheme for the matching tuples from the two relations will hash on the same node, because the *namespace* and *resourceID* will coincide. Whenever a *newData* callback is issued on the nodes in the temporary namespace saying that a tuple from *S* (or *R*) table has arrived, a *get* is called in order to search for a matching tuple from the other relation *R* (or *S*). The get function is expected to be local. Even if some failures have happened, this can be overcome at the expense of an additional round trip.
These steps are all done in parallel.
Whenever matching tuples are found, they are immediately pulled to the query initiator.

The symmetric hash join algorithm requires all matching data to be rehashed, which may need much bandwidth. *Fetch matches, Symmetrical Semi-Join Rewrite* and *Bloom Filter Rewrite* algorithms can improve *Symmetric Hash-Join* in this matter.

6. Validation and Performance Evaluation

Validation setup
The query used for simulations was the following:

```
SELECT R.pkey, S.pkey, R.pad
  FROM R, S
WHERE R.num1 = S.pkey
    AND R.num2 > constant1
    AND S.num2 > constant2
    AND f (R.num3, S.num3) > constant3
```

Tables S and R are synthetically generated. R has 10 times more tuples than S, and the attributes for R and S are uniformly distributed. The constants in the predicates are chosen to produce a selectivity of 50%. The R.pad attribute is used to insure that all result tuples are 1KB in size.

The simulator allows a scaling up to 10.000 nodes, after which it no longer fits in RAM - which results in a limitation of the simulation, not of the PIER architecture itself.
Two different topologies are used in the simulations. First, a fully connected network, where the latency between any two nodes is 100ms and the inbound link capacity of each node is 10Mbps. This is a simulation of homogeneously nodes spread throughout the Internet. Another more realistic topology is also used, but the results are qualitatively similar.
Simplifying assumptions: the focus is on the bandwidth and latency bottlenecks and computation and memory overheads of query processing are ignored. Also, data changes at a rate higher than the rate of incoming queries.
Experiments are run on a cluster of 64 PCs connected by a 1-Gbps network.

Scalability

The purpose of this test is to prove that PIER is able to scale its performance (in terms of query response time) while the number of nodes in the system increase and the workload proportionally increases.

Assume that each node is responsible for 1Mb of data.

In order to measure the response time, the authors decided to measure the time passed until the 30th tuple has arrived. They argue their choice by the following reason: if the arrival of the first item is considered, then if this item were local the experiment would not capture network limitations. They do not consider the last item either, as increasing the load and network size implies increasing the number of results for each query. This would mean then measuring a constant – the network capacity of the node where all results arrive, which is not interesting.

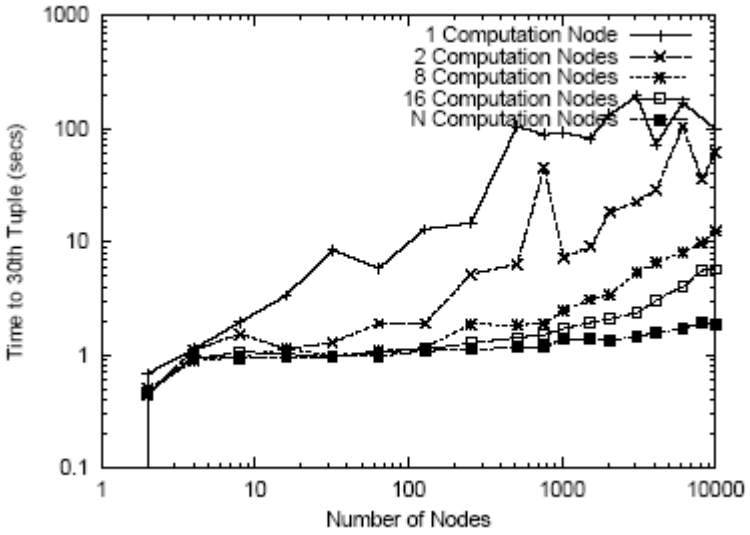

Figure 5: ***Average time to receive the 30th result tuple when both the size of the network and the load are scaled up. Each data point is averaged over three independent simulations.***

The simulations were run several times, with various numbers of computation nodes (the nodes in the temporary "queue" namespace). The number of computation nodes can be manipulated by restricting the naming scheme for the temporary namespace such that the items would be hashed onto a certain number of nodes.

If all nodes are used, the scalability results look pretty good. If the number of nodes in the network is increased 10 000 times, then the query response time increases only four times. Perfect scalability would imply the same response time. Why the four times increase? Remember that the average path length in CAN is:

$$O(n^{\frac{1}{d}})$$

The experiments used $d = 4$. Thus, an increase of a factor of 10 000 in the number of nodes result in an increase of 10 times in the average routing path length. Yet, there is some

constant time added to the overall query response time, which comes from the sending back the data to the query initiator. All these lead to the factor of 4 increase.

On the other hand, if the number of computation nodes increase, the results do not look as good as before. This is because the load per computation node increases significantly.

Effects of Soft State

As consistency has been traded off for scalability, precise measurements of the damage this implies are needed.

The parameters would be the Rate of Failures and the Average Recall (average percentage of the tuples retrieved for the given query from all tuples that match the query in the distributed database).

Some observations are needed: first, some explanations on the DHT mechanism that deals with failures.

Each node sends periodically 'keep-alive' messages to its neighbors. If it doesn't receive several consecutive answers from one of these neighbors, the node decides that neighbor has failed. Detecting a node's failure thus takes some time, which depends on the frequency of the keep-alive messages. During this time, all data sent to this node is lost. A simple algorithm to counteract this effect is for all producers of data to periodically renew their tuples. Data is rehashed from time to time.

Does this scheme work?

Simulation results prove that average recall does not decrease so dramatically with failure rate if the refresh rate is high (but this costs traffic).

The parameters of the experiments are: 4096 nodes in the network, 15 sec to detect a node failure, refresh rate varying between 30 sec and 225 sec.

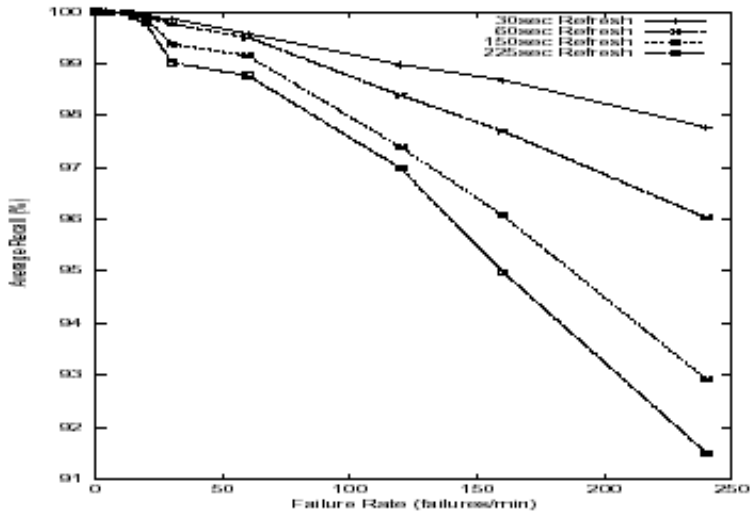

Figure 6: ***Average Recall for different refresh periods***

7. Conclusions

The paper presents the architecture of PIER – a relational query engine built on top of a DHT. The aim is the ability of running queries on Internet-scale databases.

The authors suggest a nice, possible application that could use such a query engine: Network Monitoring. The feasibility of this application is well argued, even software that could produce structured data to be used by PIER are specified.

Achieving scalability is done on the expense of relaxed consistency. The correctness of data set is redefined. Recall is measured in order to understand the prejudice of relaxed consistency. High Recall is yet obtained by frequently rehashing data. This may result in an overload of the network. The authors admit that further optimization of this algorithm is required. Better techniques of replication of DHT-based data will significantly improve the scalability.

PIER implements a particular DHT – Content Addressable Network. Its design also aims at scalability, a good, promising, sign in this direction being the fact that the size of the routing table of each node is independent on the number of nodes in the network. Still, due to the fact that the average routing path length increases with the number of nodes, the query response time increases, too. So CAN may not be the 'perfect' choice for a DHT. Chord is reminded, but no detailed results are presented, if Chord is chosen instead of CAN.

PIER has a three-tier architecture. To my opinion, it is well structured, in the sense that future improvements – such as a query optimizer – can be easily integrated.

The central topic of the paper are the distributed join algorithms. They are derived from classical, parallel joins on distributed databases [3]. One strategy is to avoid an iterator model that links operators together. Instead, results should be used as quickly as they are produced. This may hide network latency. On the other hand, all tuples that match the local join conditions are rehashed, and this operation can consume a lot of bandwidth. Improvements of the *symmetrical hash join* algorithm are proposed: *Fetch Matches*, *Symmetrical Semi-Join rewrite* and *Bloom Filter Rewrite*. These perform better with respect to query response time and bandwidth need.

Although many improvements are welcome, the architecture looks promising, from the point of view of the simulation results.

8. References

[1] Querying the Internet with PIER – Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shhenker, Ion Stoica

[2] A Scalable Content Addressable Network – Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker

[3] Dataflow Query Execution in a Parallel Main-Memory Environment – Anita N. Wilschut, Peter M.G. Apers