Querying the Internet with PIER

Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, Ion Stoica

Write up by Natalia Kozlova

Saarbrücken, 16 December 2003

Content

1 Introduction

In our post-cyber revolution world the situation is changed very quickly. And the cutting edge of research becomes broader in all directions. On the one hand, the fundamental database system issues have changed dramatically in the last decade. As such, there are ample new issues for database system research to investigate.

One can observe that there a couple of facts needed to be taken into account for databases research community [1]:

1. The Web and the Internet make it easy and attractive to put all information into cyberspace, and makes it accessible to almost everyone.
2. Ever more complex application environments have increased the need to integrate programs and data.
3. Hardware advances invalidate the assumptions and design decisions in current DBMS technology.

The Web and its associated tools have dramatically cut content creation cost, but the real revolution is that the Web has made publishing almost free. The Web is the major platform for delivery of applications and information. And the fact that hardware becomes cheaper and cheaper, will only accelerates this process.

It can be said that the Web is one huge database. However, the database research community has contributed little to the Web thus far. First, database systems are often used as high-end Web servers, as webmasters with a million pages of content invariably switch to a web site managed by database technology rather than using file system technology. Second, major content publishers are using or evaluating database systems for storing their content repositories. However, the largest of the web sites, especially those run by portal and search engine companies, have not adopted database technology. Also, smaller web sites typically use file system technology for content deployment, using static HTML pages.

Another fact is that large enterprises have hundreds, sometimes thousands, of large-scale, complex packaged and custom applications. Interoperation between these applications is essential for the flexibility needed by enterprises to introduce new web-based applications services, meet regulatory requirements, reduce time to market, reduce costs, and execute business mergers. Advances in database technology will be required to solve this application integration problem.

If technology trends continue, large organizations will have petabytes of storage managed by thousands of processors - a hundred times more processors than today. The database community is rightly proud of its success in using parallel processing for both transaction processing and data analysis. However, current techniques are not likely to scale up by two more orders of magnitude.

On the other hand we observe the Peer-to-peer (P2P) systems have recently become a very active research area, due to the popularity and widespread use of P2P systems

today, and their potential uses in future applications. It's as a way to share huge amounts of data, quite natural model for interaction between devices (e.g., via the web services).

P2P systems are popular because of the many benefits they offer: adaptation, self-organization, load-balancing, fault-tolerance, availability through massive replication, and the ability to pool together and harness large amounts of resources. File-sharing P2P systems distribute the main cost of sharing data - bandwidth and storage - across all the peers in the network, thereby allowing them to scale without the need for powerful, expensive servers.

However, P2P systems also present several challenges that are currently obstacles to their widespread acceptance and usage - e.g., security, efficiency, and performance guarantees like atomicity and transactional semantics. The P2P environment is particularly challenging to work in because of the scale of the network and unreliable nature of peers characterizing most P2P systems today. Many techniques previously developed for distributed systems of tens or hundreds of servers may no longer apply; new techniques are needed to meet these challenges in P2P systems. And one of the main problems is search. Though data-sharing P2P systems are capable of sharing enormous amounts of data, such a collection is useless without a search mechanism allowing users to quickly locate a desired piece of data.

There are a number of approaches to incorporate search mechanisms in the existing P2P system structure. Some of them use centralized index, but this architecture cannot exploit the distributed nature of P2P systems efficiently. There is an obvious need in techniques that provide an efficient distributed search mechanism with more advanced functionality than simple keyword search.

An important area of research therefore lies in developing mechanisms for richer query languages. As a complex language defined over a rich data model, SQL is the most difficult query language to support among the examples listed. Current research on supporting SQL in P2P systems is very preliminary. The PIER project supports a subset of SQL over a P2P framework.

Thus we came to the main goal of this work – to describe PIER and its features. PIER is a distributed query engine based on overlay networks that can work on various data. It is intended to bring database query processing facilities to new, widely distributed environments. PIER tries to acquire advantages from both of its predecessors: P2P systems bring scalability and flexibility when DB systems bring relational model and query facilities.

2 Design Principles

There are a couple of principles, which guided PIER's developers in their intention to provide scalable architecture.

2.1 Relaxed Consistency

When talking about modern distributed systems, there can be pointed three criteria we need to balance between. They are Consistency, Availability and Partition tolerance or CAP for short [3].

- Consistency: there has been significant research designing ACID databases, and most of the new frameworks for building distributed web services depend on these databases. Interactions are expected to behave in a transactional manner.
- Availability: every request should succeed and receive a response. When a service goes down, it may well create significant real-world problems. The goal is to be as available as the network on which they run: if any service on the network is available, then the service should be accessible.
- Partition tolerance: when some nodes crash or some communication links fail, it is important that the service still perform as expected on the survived (or reachable) part of the network.

It was noticed, that modern systems usually choose consistency and prefer partition tolerance to availability. By contrast, PIER developers want their system to become part of the “integral fabric” of the Internet – thus it must be highly available, and work on whatever subset of the network is reachable. In the absence of transactional consistency, PIER will have to provide best-effort results.

2.2 Organic Scaling

The main idea here - the system should scale in a natural way when the number of nodes changed. And it means that we should not limit ourselves with data, allocated in the certain place. Data should be decentralized, distributed across nodes, participated in the network. This also leads us to the cheaper architecture we discuss in the first part of this work – to support this system we have no need to buy expensive servers or wide Internet channel. As long as data is distributed, the system will organically scale when nodes join and leave.

2.3 Natural Habitats for Data

One of the main barriers for distributing databases among usual users is the need to load data they owned into database. And database have a schema thus you data can be modified somehow to be available. Next, you can access data in the database only via

DB interface or certain tools. And this is a serious limitation, because of the very heterogeneous nature of the Internet. PIER allows data to be heterogeneous – file system or live feed. Hence the business of "wrappers" is to provide necessary information for query engine.

2.4 Standard Schemas via Grassroots Software

This principle leads us to the idea: why should we invent various tricky methods to integrate data of the different users if we are already have many programs spread so wide, that they become a de facto standards? For example, one of the challenging applications of the PIER is network monitoring, and here you can find such programs as TBIT or tcpdump. And the schemas they provide are well-known and wide used. The ability to stitch local analysis tools and reporting mechanisms into a shared global monitoring facility is both semantically feasible and extremely desirable. This is also utilized by PIER.

3 PIER Architecture

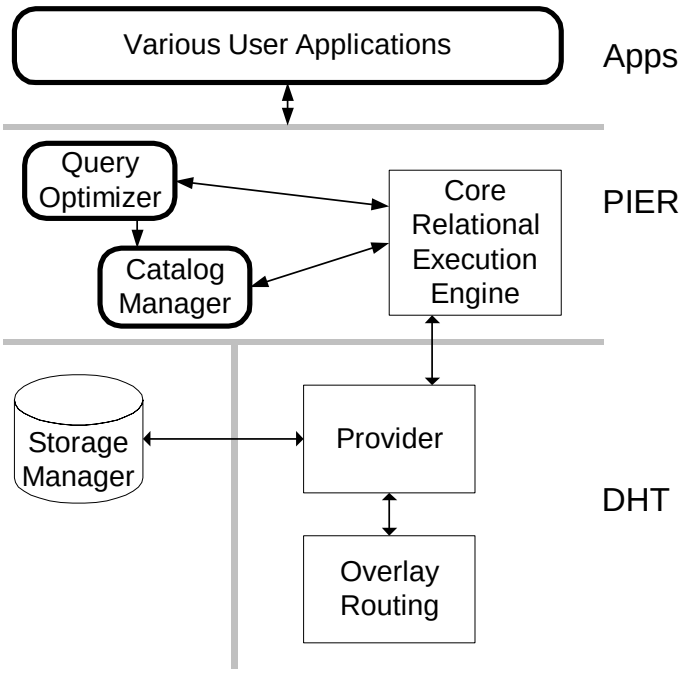

Figure 1. PIER Architecture

As far as we discussed the motivation and the design principles, we will continue by describing PIER's architecture. On the picture you can see the structure of the system.

As we discuss, PIER is query engine, not a Database system. And we don't focus on storage and transactions, but just querying existing data.

PIER is a three-tier system as shown in Figure 1. Applications interact with the PIER Query Processor (QP),

which utilizes an underlying DHT.

An instance of each DHT and PIER component is run on each participating node.

The next explanation will be organized as follows: first we'll study the lowest DHT layer; then we will go to the Query processor which is PIER itself, next section will be devoted to the performance evaluation; we will talk a little about the applications and after all I will summarize the topics, described above.

4 DHT Implementation

4.1 CAN design

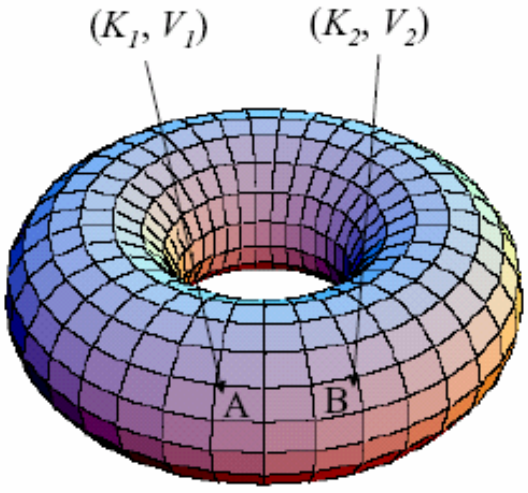

Figure 2. One of the possible CAN spaces

Before we will go into details I want to see a few words about DHTs in general. DHT provides a hash table abstraction over multiple distributed compute nodes. Each node in a DHT can store data items, and each data item is identified by a unique *key*. At the heart of a DHT is an *overlay routing* scheme that delivers requests for a given key to the node currently responsible for that key. This is done without any global knowledge – or permanent assignment – of the mapping of keys to machines. Routing proceeds in a multi-hop fashion; each node maintains only a small set of neighbors, and routes messages to the neighbor that is in some sense "nearest" to the correct destination.

In particular, this implementation of PIER based on the CAN, a DHT architecture described in [4]. I will not explain here in details how CAN works, just a little description. It should be said that PIER can also use another DHT architectures.

CAN based on d-dimensional Cartesian coordinate space. Space is partitioned into zones – hyper-rectangles. Each node owns a distinct zone, maintains routing table with neighbors. Uniform hash function maps a key onto a point, on the Figure 2 there is a key K1 mapped to example zone P1. Pair (*K1*, *V1*) is stored at the node *A*, which owned the zone with *P1*. We can issue Insertion, lookup, and deletion of (*key*, *value*) pairs. Lookup processes by forwarding the message along a path that approximates the

straight line in the coordinate space from the sender to the node storing the key; here it will be the shortest line form *A* to *B*, if *A* is interested in V2.

The average number of the hops for message to get from point to point is $\frac{d}{4} n^{\frac{1}{4}}$ where *d* is the number of dimensions and *n* is the number of nodes. In PIER *d* = 4.

4.2 DHT layers

Remember Figure 1. Here we will discuss the details of the lowest layer implementation.

The first layer is the routing layer and its business is to map a key into the IP address of the node currently responsible for that key. The API for this layer is simple; it's shown on Figure 3.

```
lookup(key) → ipaddr
join(landmarkNode)
leave()
locationMapChange
```

Figure 3. Routing layer API

`lookup` is an asynchronous function which will issue a callback when the node has been located. The `locationMapChange` callback is provided to notify higher levels asynchronously when the set of keys mapped locally has changed. The `join` and `leave` calls provide for creating a new overlay network, attaching to an existing network, and gracefully leaving the network. For preexisting networks, the `join` method requires the socket address of any node already in the network (or NULL to start a new network).

The next part of the DHT is the Storage Manager. Its task is to store and retrieve records, which consist of key/value pairs. Keys are used to locate items and can be any data type or structure supported. This means that you are not limited in you choice of storing data – it can be main memory structures or indexing systems like Berkley DB or even a file system. PIER uses main memory structures for simplicity now. API is shown on Figure 4. We can store a key in DHT using `store`, obtain an item by a key using `retrieve` and remove and item from DHT table issuing `remove`.

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

Figure 4. Storage Manager API

The last part of DHT we need to discuss is Provider. Surprisingly, it provides an interface to higher levels and manages routing and storage tasks using proper components.

Each object in DHT characterized with *namespace*, *resourceID*, and *instanceID*. We apply a hash function to namespace, resourceID to obtain DHT key. The namespace

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

Figure 5. Provider API

identifies the application or group an object belongs to; for query processing each namespace corresponds to a relation. Namespaces do not need to be predefined.

When the namespace is what object belongs to, the resourceID is generally intended to be a value that can tell what is object itself, some identifier. Usually it can be the primary key for base tuples, although any attribute (or combination) could be used for this purpose. The instanceID is an integer randomly assigned by the user application, which serves to allow the storage manager to separate items with the same namespace and resourceID (which can occur when items are not stored based on the primary key). The API is shown on the Figure 5.

`get` issued to obtain an item, `put` called to insert an item into DHT. Lifetime parameter is necessary to tell DHT during which time should it store this object, this is adherence to principle of "relaxed consistency". And when this time will pass, the item will be deleted from DHT. Thus node that owns data should periodically call `renew` to show that it is alive and this item is up to date.

The next function, multicast addresses the propagation of information between particular nodes. For details see [5]. A multicast message contains the location prefix (corresponding to the namespace), mask, and a payload, which is the data to be delivered to each node. Using the prefix and mask, the lower and upper bounds of the location identifier space can be determined. A multicast message is sent to a *multicast zone*, described by two coordinates (the lower and upper bounds of the multicast range), in the logical space where all identifiers of interest are mapped. This zone can intersect one or more nodes.

Flooding of the multicast message begins by the message being delivered to any node in the multicast zone. This can be achieved by issuing a lookup for a random identifier within the multicast range. There are some algorithms, but we more interested in general idea. Each node iterates through its neighbors and decides whether to forward the message. This decision based on the fact if the neighbor is not the node the message came from and if the neighbor's zone intersects with the multicast zone.

The fifth function from Figure 5 is `lscan`. Each node calls `lscan` to iterate through its local data. And the last function, `newdata,` is a callback issued when the data, stored on a particular node changed. This callback addressed to the all nodes in its namespace.

5 Query Processor

Let's return back to the Figure 1. The next layer on the top of DHT is PIER itself. It also consists of three components but it's a kind of idea. Now the upper level contains only query engine… but it's enough for the moment!

The PIER Query Processor is a "boxes-and-arrows" dataflow engine, supporting the simultaneous execution of multiple operators that can be pipelined together to form traditional query plans. In our initial prototype, we started by implementing operators for selection, projection, distributed joins, grouping, and aggregation.

It's important for distributed systems to hide somehow network latencies, so in PIER operators produce results as quickly as possible (push) and enqueue the data for the next operator (pull).

The next task is modification of data. But as you remember here we always emphasize the fact that data is heterogeneous. And in fact the data is stored separately from the query engine; it is not a standard DB practice. PIER is not a Database, and the difference is we don't focus on storage and transactions, just querying existing data. Modification performed using functions of DHT-level Provider.

And the last but not less important question: OK, we have a network, QP and data. But *what* data should we choose to push into selection procedure? In P2P networks there are usual situations when some of nodes failed. It can be fatal for usual DB, but here the "relaxed consistency" principle was claimed.

PIER defines *reachable snapshot* to select data from nodes that are alive at the time the query was sent. But there are problems with network latency and time synchronization so *dilated reachable snapshot* defines data which are available at the time the query arrives at the reachable nodes.

This is a short description of the PIERs QP. And now I want to show you most interesting things from this outline in details. We will talk about joins – the core of query processing.

These join algorithms are adaptations known ones, but with leveraging DHTs whenever possible. PIER uses DHTs in both of the senses used in the literature – as "content-addressable networks" for routing tuples by value, and as hash tables for storing tuples. DHTs provide hash indexes and the hash tables that underlie many parallel join algorithms and the main point here that the set of nodes these algorithms running on is volatile. This was not available in earlier database.

5.1 Core join algorithms

Imagine that we have two relations *R* and *S*, distributed in the network and we want to join them on some attributes. The related namespaces are N_R and N_S. It should be

mentioned, that of course, data is already hashed by resourceID in the hash table so there we are talking about rehashing. And this rehashing performed by join attributes.

The first algorithm is **symmetric hash join**.

It works as follows:

- Someone initiates query, query is `multicasted` to the nodes in N_R and N_S.
- Each node starts to `lscan` its own piece of relation (Figure 6 a).
- When the tuple, that satisfy join condition is found (in R or S), it must be copied in the new temporary namespace for join result. It copies only necessary columns. The resourceID of this copy is hashed concatenation of its join attributes. To distinguish tuples they are tagged with their source table name. In other words, when tuple is found, we rehash it by join attributes and call `put` to insert it into namespace (say N_Q) defined by hash function (Figure 6 b).
- When the tuple is `put` into N_Q node, responsible for the zone, it's put in issues `newData` to notify that it has a new candidate to join (say from table *R*).
- Each node in N_Q calls `get` to find matches for this tuple (from *R*) in *Q* from another table (*S*).
- Matches are joined and sent to the next stage of the query (if exist) or to initiator.

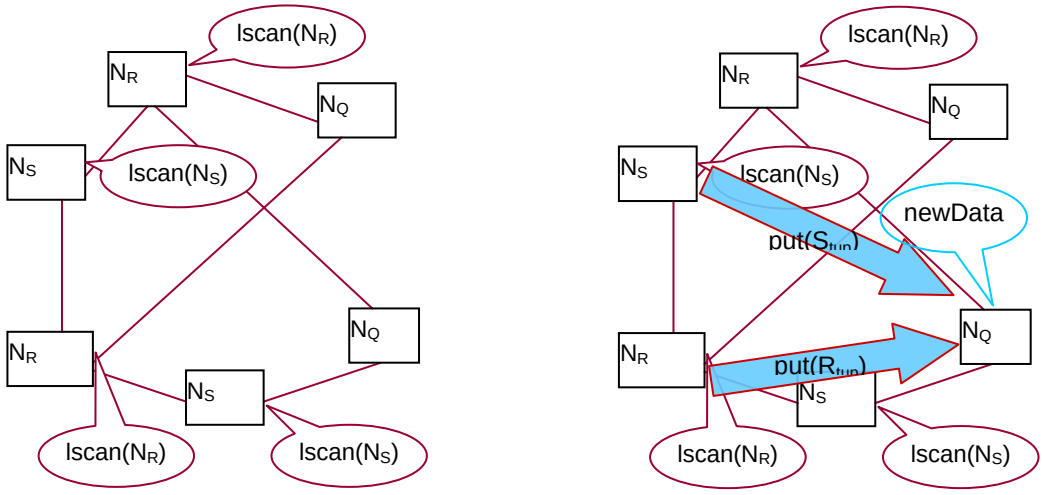

Figure 6. SHJ

This algorithm send projected tuples matching some predicates, but can't pre-check join predicate. It requires a lot of rehashing and produces high traffic.

The second algorithm is **fetch matches**. It works when one of the tables (say *S*) is already hashed on the join attributes.

It works as follows:

- Each node in opposite table *R* starts to `lscan` its own piece of relation.
- When the tuple, that satisfy join condition is found (in R), node issues `get` for corresponding *S* tuple (Figure 7 a).
- `get` returns matched tuple(s), they are joined with *R* tuple (Figure 7 b).
- Matches are joined and sent to the next stage of the query (if exist) or to initiator.

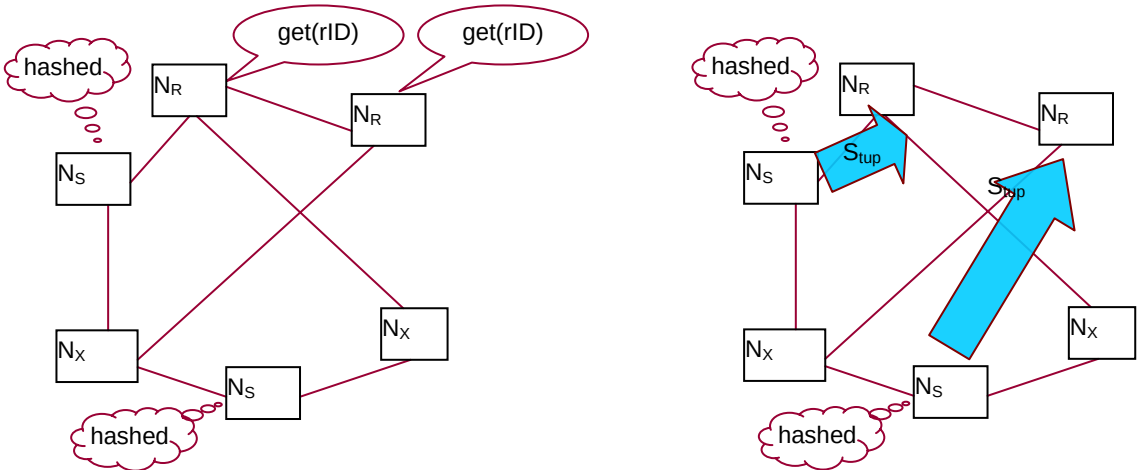

Figure 7. Fetch Matches

This algorithm only retrieve tuples matching join predicate, but entire tuple must be fetched and may not match some predicates.

5.2 Joins rewriting.

As we noticed, previous two algorithms, especially first one requires a lot of network resources. In real network there is no guarantee that you can consume as much as you want. Thus there is an obvious need in less traffic-consuming algorithms.

Well, there is another algorithm, called **symmetric semi-join**. It exploits the first two to produce a better results (we are speaking about traffic) in a price of latency in obtaining final result. It works as follows: first locally project both *R* and *S* to their *resourceID*s and join keys, and perform a SHJ on the two projections. The resulting tuples are pipelined into FM joins on each of the tables' *resourceID*s .Doing this we minimize initial communication.

The second rewriting strategy is to use **Bloom Filters**.

Abstractly a Bloom filter is a mutable object that maps strings onto bits. Given a string it returns a bit. It is allowed false positives. A creation time parameter trades off the frequency of false positives against storage costs. A bloom filter can also have strings added which causes the filter to respond true to that string in the future. Several Bloom filters can be OR-ed together. The merged filter will respond positively at least whenever each of the inputs would have.

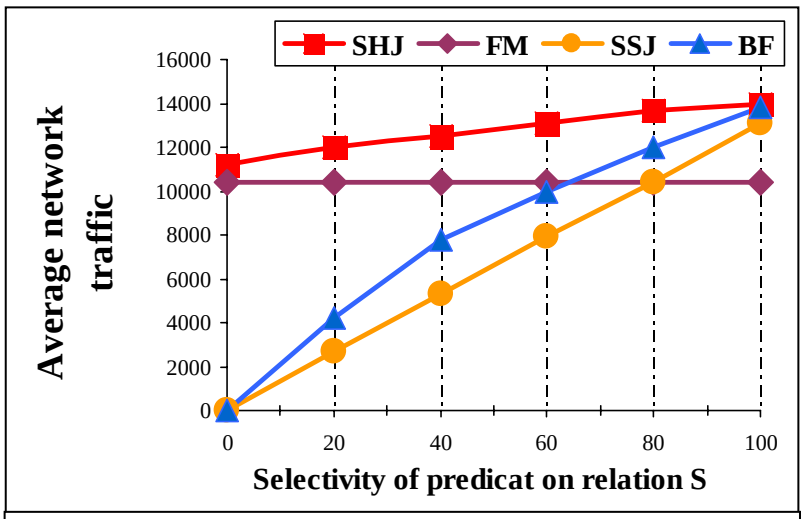

Figure 8. Aggregate network traffic for each strategy

The Bloom join works as follows: at first each node creates Bloom Filter for its *R* and *S* fragments and publish it into a temporary DHT namespace for each table. Filters are OR-ed together and then `multicasted` to all nodes storing the opposite table. Node `lscans` corresponding table fragment, but rehashing only tuples matched the filter. This strategy reduces rehashing.

5.3 Grouping/Aggregation

Here I briefly describe the Grouping/Aggregation techniques. These techniques are not presented in the source paper; however I want just to show that the method they use is similar.

To perform aggregation operation, each node:

- `lscan` it's fragment of the source table and determine the group tuple belongs in
- adds tuple's data to that group's partial summary
- for each group represented at the site, rehash the summary tuple with hash key based on group-by attribute. `put` tuple into temporary grouping namespace.
- nodes in grouping namespace use `newData` to get partial summaries, combine and produce final result

5.4 Performance evaluating

After we discussed join algorithms, it's time to look on the evaluating of performance. In simulation setup inbound capacity of the node is equal to 10 mbps. Size of relations *R* plus *S* is 25 GB. System has 1024 nodes.

SHJ	Fetch Matches	symmetric semi-join	Bloom Filter
3.73 sec	3.78 sec	4.47 sec	6.85 sec

Table 1. Average time to receive last tuple

On Figure 8 you can see the bandwidth requirements for each strategy as a function of the selectivity of the predicate on *S*. As you can see, the symmetric hash join uses the most network resources since both tables are rehashed. The number of results increases with the selectivity of the selection on S.

Fetch Matches strategy basically uses a constant amount of network resources because the selection on S cannot be pushed down in the query plan. This means that regardless of how selective the predicate is, the S tuple must still be retrieved and then evaluated against the predicate at the computation node.

In the symmetric semi-join rewrite, the second join transfers only those tuples of S and R that match. As a result, the total inbound traffic increases linearly with the selectivity of the predicate on S.

In the Bloom Filter case, as long as the selection on S has low selectivity, the Bloom Filters are able to significantly reduce the rehashing on R, as many R tuples will not have an S tuple to join with. However, as the selectivity of the selection on S increases, the Bloom Filters are no longer effective in eliminating the rehashing of R tuples, and the algorithm starts to perform similar to the symmetric join algorithm.

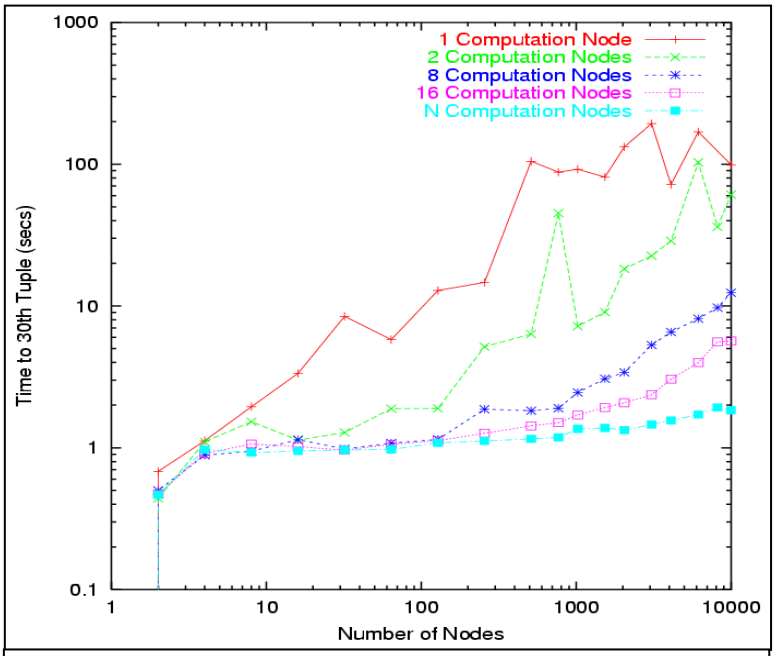

Figure 9. Time for the 30th tuple

This algorithms show similar trend during measurement of the time, needed to receive last tuple. Thus there is no real need to show the full plot. In the table you can see the average time to receive last tuple.

5.5 Scalability simulation

The main idea of the PIER is that it is distributed query processor that scales well. Let's study how it scales. The conditions are: |R| =10 |S, constants produce selectivity of 50%, each node is responsible for 1 MB of source data. On the Figure 9 you can see the response time for the 30th tuple. This value was chosen to be after the first tuple received and well before the last.

The number of overlay hops for each lookup increases as the network size increases, this leads to an increase in the lookup latency. In CAN the lookup length increases in $n^{0,25}$ with d = 4.

When the number of computation nodes is kept small by constraining the join namespace N_Q the bottleneck moves to the inbound links of the computation nodes, and as a result the performance of the system degrades significantly as the total number of nodes and therefore the load per computation node increases.

In summary, PIER scales well as long as the number of computation nodes is large enough to avoid network congestion at those nodes.

The query, used in evaluation, is:

```
SELECT R.key, S.key, R.pad
FROM R,S
WHERE R.n1 = S.key
      AND R.n2 > const1
      AND S.n2 > const2
      AND f(R.n3,S.n3) >  const3
```

At Figure 10 you can see the plot with experimental results of running prototype implementation on a cluster of 64 PCs connected by an 1 Gbps network. It is also a time to receive 30th tuple when number of nodes scales from 2 to 64.

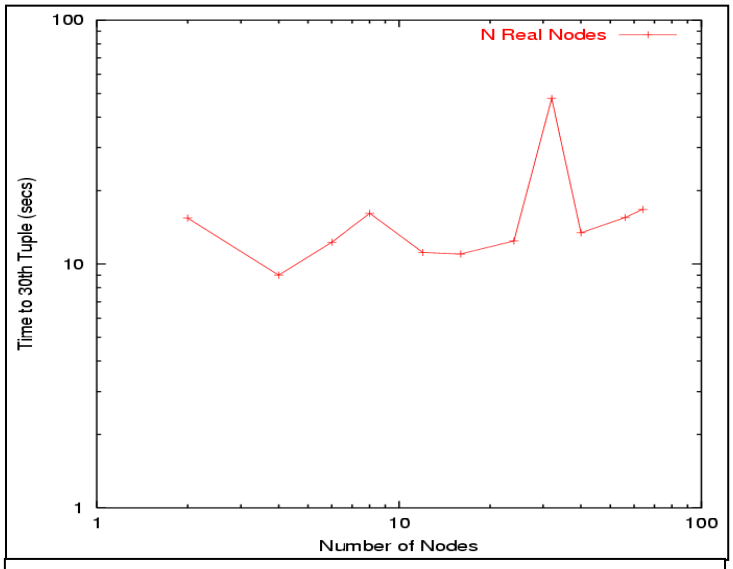

Figure 10. Time for the 30th tuple

As expected the time to receive the 30-th result tuple practically remains unchanged as both the system size and load are scaled up. The reason that the plot is not smooth is because the cluster we used to run our experiments was typically shared with other competing applications, and was particularly heavily loaded during the period we ran these tests. Developers believe the peak in response time at 32 nodes is due to an artifact in their CAN implementation.

6 Conclusion

Well, we discussed PIER - a structured query system intended to run at large scale. Due to its design, it can perform queries on heterogeneous data. The scalability of PIER derives from a small set of relaxed design principles, which led to some of key decisions, including: the adoption of soft state and dilated-reachable snapshot semantics; use of DHTs as a core scalability mechanism for indexing, routing and query state management.

We should not take into account that presented system is the first draft and requires a lot of work. There are some common issues:

- Caching – Both at DHT and QP levels
- Using Replication – for speed and fault tolerance (both in data and computation)
- Security

And also there are some related to databases issues:

- Pre-computation of (intermediate) results
- Continuous queries/alerters
- Query optimization (Is this like network routing?)
- More algorithms, Dist-DBMS have more tricks
- Performance Metrics for P2P QP Systems

7 Literature

Main paper: **Querying the Internet with PIER (2003)** Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham Boon, Thau Loo, Scott Shenker, Ion Stoica. Proceedings of 19th International Conference on Very Large Databases (VLDB) http://citeseer.nj.nec.com/huebsch03querying.html

1. The Asilomar Report on Database Research. Phil Bernstein, Michael Brodie, Stefano Ceri et al.

2. Open Problems in Data-Sharing Peer-to-Peer Systems. Neil Daswani, Hector Garcia-Molina, and Beverly Yang

3. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. ACM SIGACT News, 33(2), June 2002.

4. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In Proc. 2001 ACM SIGCOM Conference, Berkeley, CA, August 2001.

5. R. Huebsch. Content-based multicast: Comparison of implementation options. Technical Report UCB/CSD-03-1229, UC Berkeley, Feb. 2003.

6. M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. In *1st International Workshop on Peer-to-Peer Systems (IPTPS'02)*, March 2002.