Querying the Internet with PIER

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

EECS Computer Science Division, UC Berkeley
{huebsch@,jmh@,nickl@db.,boonloo@,istoica@}cs.berkeley.edu,

*International Computer Science Institute
shenker@icsi.berkeley.edu

Abstract

The database research community prides itself on scalable technologies. Yet database systems traditionally do not excel on one important scalability dimension: the degree of distribution. This limitation has hampered the impact of database technologies on massively distributed systems like the Internet.

In this paper, we present the initial design of PIER, a massively distributed query engine based on overlay networks, which is intended to bring database query processing facilities to new, widely distributed environments. We motivate the need for massively distributed queries, and argue for a relaxation of certain traditional database research goals in the pursuit of scalability and widespread adoption. We present simulation results showing PIER gracefully running relational queries across thousands of machines, and show results from the same software base in actual deployment on a large experimental cluster.

1 Introduction

The database research community prides itself on the scalability of its technologies. The challenge of supporting "Very Large Data Bases" is core to the community's identity, and ongoing research on scalability has continuously moved the field forward. Yet database systems do not excel on one important scalability dimension: the degree of distribution. This is the key scalability metric for global networked systems like the Internet, which was recently estimated at over 162 million nodes [9]. By contrast, the largest database systems in the world scale up to at most a few hundred nodes. This surprising lack of scalability may help explain the database community's lament that its technology has not become "an integral part of the fabric" of massively distributed systems like the Internet [4].

In this paper, we present PIER (which stands for "Peer-to-Peer Information Exchange and Retrieval"), a query engine that comfortably scales up to thousands of participating nodes. PIER is built on top of a Distributed Hash Table (DHT), a peer-to-peer inspired overlay network technology that has been the subject of much recent work in the networking and OS communities [26, 31, 29, 36]. We present simulation results showing PIER gracefully running relational queries across thousands of machines, and initial empirical results of the same software base in actual deployment on our department's largest active cluster of computers.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

Our agenda in this initial paper on PIER is twofold. We show that a system like PIER presents a "technology push" toward viable, massively distributed query processing at a significantly larger scale than previously demonstrated. In addition, we present what we believe to be an important, viable "application pull" for massive distribution: the querying of Internet-based data *in situ*, without the need for database design, maintenance or integration. As a team of network and database researchers, we believe in the need and feasibility of such technology in arenas like network monitoring.

The primary technical contributions of this work are architectural and evaluative, rather than algorithmic. We argue that certain standard database system design requirements are best relaxed in order to achieve extreme scalability. We present a novel architecture marrying traditional database query processing with recent peer-to-peer networking technologies, and we provide a detailed performance study demonstrating the need for and feasibility of our design. Finally, we describe how our architecture and tradeoffs raise a number of new research questions, both architectural and algorithmic, that are ripe for further exploration.

2 Querying the Internet

In this section, we present some motivating applications for massively distributed database functionality, and from them extract design principles for a reusable system to address these applications.

2.1 Applications and Design Principles

Peer-to-peer (P2P) filesharing is probably the best-known Internet-scale query application today – in today's post-Napster, post-AudioGalaxy era, these tools truly run queries across the Internet, and not on centralized servers. The reason is not particularly noble: decentralized data spreads responsibility and tracking of copyright violation, motivating *in situ* processing of file data and metadata (filenames, sizes, ID3 tags, etc.) The wide distribution of this data comes from massive deployment by "normal", non-expert users. These systems are not perfect, but they are very useful, and widely used. In this sense, they echo the rise of the Web, but their query functionality is richer than point-to-point HTTP, and "closer to home" for database research.

We believe there are many more natural (and legal) applications for *in situ* distributed querying, where data is generated in a standard way in many locations, and is not amenable to centralized, "warehouse"-type solutions. Warehousing can be unattractive for many reasons. First, warehouses are best suited for historical analysis; some applications prefer live data. Second, warehouses can be expensive to administer, requiring a data center with sufficient storage and bandwidth to scale. Finally, socio-political concerns may discourage warehouses – with their centralized control and responsibility – when distributed query solutions are available.

An application category of particular interest to us is widespread *network monitoring*. Network protocols like IP, SMTP, and HTTP tend to have standard data representations, and also have widespread server implementations that could plausibly participate in "wrapping" the data for useful distributed query processing.

As a concrete example, we discuss the problem of network intrusion detection. Network behaviors can be categorized by "fingerprints", which may be based on many kinds of data: sequences of port accesses (e.g., to detect port scanners), port numbers and packet contents (for buffer-overrun attacks or web robots,) or application-level information on content (for email spam). An intrusion is often hard to detect quickly except by comparing its "fingerprint" to other recently experienced attacks. We believe that many standard network servers (e.g., mail servers, web servers, remote shells, etc.) and applications (e.g., mail clients, calendar programs, web browsers, etc.) could bundle fingerprint-generating wrappers to optionally participate in distributed intrusion detection.

PIER provides a way to flexibly and scalably share and query this fingerprint information. Each attacked node can publish the fingerprint of each attack into PIER's distributed index, where it will persist for some period before "aging out" (Section 3.2.3). To determine the threat level, organizations can then periodically query the system to see which fingerprints are similar, how many reports exist, etc.

For example, in order to find compromised nodes on the network, it may be useful to use multiple fingerprint-wrappers to identify multiple kinds of "intrusions" from a single domain – e.g., to identify unrestricted email gateways (often a channel for spam) running in the same subnet as a web robot (which may be crawling for email addresses):

```
SELECT S.source
  FROM spamGateways AS S, robots AS R
 WHERE S.smtpGWDomain
        = R.clientDomain;
```

In a more general environment, a summary of widespread attacks can be a simple aggregation query over a single fingerprint table:

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

Organizations may treat some reporters as more useful or reliable than others, and therefore may want to weigh results according to their own stored judgment of reputations. This can be easily accomplished with the following query:

```
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;
```

We choose intrusion detection examples because we expect many people would willingly set their servers to "opt in" to a communal system to improve security[1]. However, many analogous examples can be constructed using standard network tools available today. For example, network tools like `tcpdump` can be used to generate traces of packet headers, supporting queries on bandwidth utilization by source, by port, etc. Beyond the analysis of packet headers (which are arguably "metadata"), intrusion detection tools like Snort [28] can take a packet stream and generate signatures much like those described above, by examining both packet headers and the data "payloads" that they carry. Tools like TBIT [24] can be used to support queries about the deployment of different software versions (TBIT reports on TCP implementations); this can be useful for doing "public health" risk assessment and treatment planning when security holes are identified in certain software packages. Query applications outside of networking are also plausible, including resource discovery, deep web crawling and searching, text search, etc.

We intend for PIER to be a flexible framework for a wide variety of applications – especially in experimental settings where the development and tuning of an application-specific system is not yet merited. We are interested in both the design and utility of such a general-purpose system – our goal is both to develop and improve PIER, and to use it for networking research.

2.2 Relaxed Design Principles for Scaling

The notion of a database system carries with it a number of traditional assumptions that present significant, perhaps insurmountable barriers to massive distribution. A key to the scalability of PIER is our willingness to relax our adherence to database tradition in order to overcome these barriers. Based on the discussion above, we identify four design principles that will guide our attempt to scale significantly:

a) Relaxed Consistency

While transactional consistency is a cornerstone of database functionality, conventional wisdom states that ACID transactions severely limit the scalability and availability of distributed databases. ACID transactions are certainly not used in any massively distributed systems on the Internet today. Brewer neatly codifies the issue in his "CAP Conjecture" [11] which states that a distributed data system can enjoy only two out of three of the following properties: Consistency, Availability, and tolerance of network Partitions. He notes that distributed databases always choose "C", and sacrifice "A" in the face of "P". By contrast, we want our 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, we will have to provide best-effort results, and measure them using looser notions of correctness, e.g., precision and recall.

b) Organic Scaling

Like most Internet applications, we want our system's scalability to grow organically with the degree of deployment; this degree will vary over time, and differ across applications of the underlying technology. This means that we must avoid

[1]The sharing of such data can be made even more attractive by integrating anonymization technologies, as surveyed in e.g. [7].

architectures that require *a priori* allocation of a data center, and financial plans to equip and staff such a facility. The need for organic scaling is where we intersect with the current enthusiasm for P2P systems. We do not specifically target the usual P2P environment of end-user PCs connected by modems, but we do believe that any widely distributed technology – even if it is intended to run on fairly robust gateway machines – needs to scale in this organic fashion.

c) Natural Habitats for Data

One main barrier to the widespread use of databases is the need to load data into a database, where it can only be accessed via database interfaces and tools. For widespread adoption, we require data to remain in its "natural habitat" – typically a file system, or perhaps a live feed from a process. "Wrappers" or "gateways" must be provided to extract information from the data for use by a structured query system. While this extracted information may be temporarily copied into the query system's storage space, the data of record must be expected to remain in its natural habitat.

d) Standard Schemas via Grassroots Software

An additional challenge to the use of databases – or even structured data wrappers – is the need for thousands of users to design *and integrate* their disparate schemas. These are daunting semantic problems, and could easily prevent average users from adopting database technology – again frustrating the database community's hopes of being woven into the fabric of the Internet. Certainly networking researchers would like to sidestep these issues! Fortunately, there is a quite natural pathway for structured queries to "infect" Internet technology: the information produced by popular software. As argued above, local network monitoring tools like Snort, TBIT and even tcpdump provide ready-made "schemas", and – by nature of being relatively widespread – are *de facto* standards. Moreover, thousands or millions of users deploy copies of the same application and server software packages, and one might expect that such software will become increasingly open about reporting its properties – especially in the wake of events like the "SQL Slammer" (Sapphire) attack in January, 2003. The ability to stitch local analysis tools and reporting mechanisms into a shared global monitoring facility is both semantically feasible and extremely desirable.

Of course we do not suggest that research on widespread (peer-to-peer) schema design and data integration is incompatible with our research agenda; on the contrary, solutions to these challenges only increase the potential impact of our work. However, we do argue that massively distributed database research can and should proceed without waiting for breakthroughs on the semantic front.

3 PIER Architecture

Given this motivation, we present our design, implementation and study of PIER, a database-style query engine intended for querying the Internet. 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.

3.1 Distributed Hash Tables (DHTs)

The term "DHT" is a catch-all for a set of schemes sharing certain design goals ([26, 31, 29, 36], etc.); we will see an example shortly in Section 3.1.1. As the name implies, a 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.

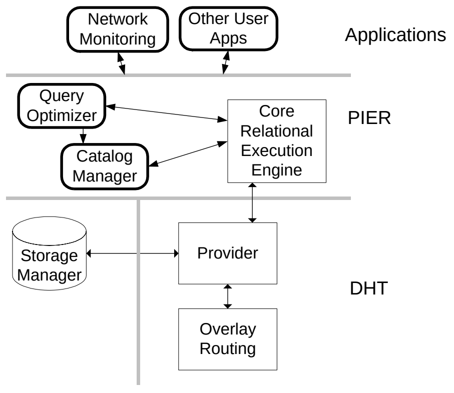

Figure 1: PIER Architecture

DHTs provide strong theoretical bounds on both the number of hops required to route a key request to the correct destination, and the number of maintenance messages required to manage the arrival or departure of a node from the network. By contrast, early work on P2P routing used "unstructured", heuristic schemes like those of Gnutella and KaZaA, which provide no such guarantees: they can have high routing costs, or even fail to locate a key that is indeed available somewhere in the network.

In addition to having attractive formal properties, DHTs are becoming increasingly practical for serious use. They have received intense engineering scrutiny recently, with significant effort expended to make the theoretical designs practical and robust.

3.1.1 Content Addressable Network (CAN)

PIER currently implements a particular realization of DHTs, called a Content Addressable Network [26]. CAN is based on a logical d-dimensional Cartesian coordinate space, which is partitioned into hyper-rectangles, called zones. Each node in the system is responsible for a zone, and a node is identified by the boundaries of its zone. A key is hashed to a point in the coordinate space, and it is stored at the node whose zone contains the point's coordinates[2]. Figure 2(a) shows a 2-dimensional $[0,16] \times [0,16]$ CAN with five nodes.

Each node maintains a routing table of all its neighbors in the coordinate space. Two nodes are neighbors in a plane if their zones share a hyper-plane with dimension d-1. The lookup operation is implemented 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.

[2]To map a unidimensional key into the CAN identifier space, we typically use d separate hash functions, one for each CAN dimension.

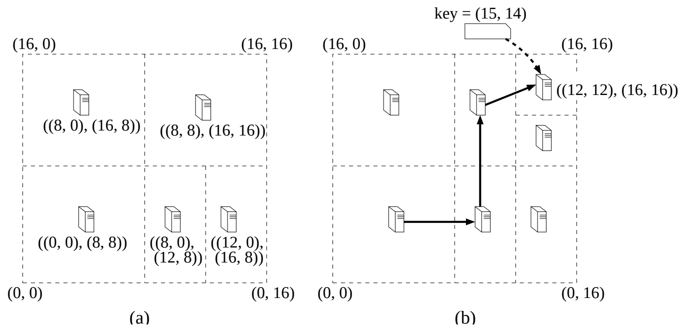

Figure 2: (a) A 2-dimensional $[0, 16] \times [0, 16]$ CAN with five nodes; the zone owned by each node is represented by its bottom-left and the top-right coordinates. (b) An example of a lookup for key $(15, 14)$ initiated by node owning the zone $((0, 0), (8, 8))$.

`lookup(key)` $\rightarrow$ `ipaddr`
`join(landmark)`
`leave()`
`locationMapChange()`

Table 1: Routing Layer API

Figure 2(b) shows the path followed by the lookup for key $(15, 14)$. Each node maintains $O(d)$ state, and the average number of hops traversed by a message is $\frac{d}{4}n^{1/d}$, where n is the number of nodes in the system (see [26] for details). We have chosen $d = 4$ in our simulations and experiments, leading to a growth behavior of $n^{\frac{1}{4}}$. However, this growth can be reduced to logarithmic by setting $d = \log n$ or using a different DHT design [31, 29, 36]. Thus, the scalability results we show here could be improved yet further via another DHT.

3.2 DHT Design

There is active debate within the DHT research community as to how to best factor DHT functionality into subsidiary components. In this paper we have chosen one particular split, which we present here. We expect that we can adapt our design to "conventional wisdom" as it accrues in the DHT community. As a validation exercise, we also deployed PIER over a competing DHT design called Chord [31], which required a fairly minimal integration effort.

3.2.1 Routing Layer

As mentioned in Section 3.1, the core of the DHT is the dynamic content routing layer, which maps a key into the IP address of the node currently responsible for that key. The API for this layer is small and simple, providing just three functions and one callback as shown in Table 1.

`lookup` is an asynchronous function which will issue a callback when the node has been located[3]. The mapping between keys and nodes is constantly changing as nodes enter and leave the network. 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 pre-existing networks, the `join` method simply requires the socket address of any node already in the network (or `NULL` to start a new network). For popular networks, it is assumed that there would be a list of landmarks in a well known location (a la `www.napigator.com`).

[3]If the key maps to the local node, then the `lookup` call is synchronous, returning true immediately with no callback.

`store(key, item)`
`retrieve (key)` $\rightarrow$ `item`
`remove(key)`

Table 2: Storage Manager API

`get(namespace, resourceID)` $\rightarrow$ `item`
`put(namespace, resourceID, instanceID, item, lifetime)`
`renew(namespace, resourceID, instanceID, item, lifetime)` $\rightarrow$ `bool`
`multicast(namespace, resourceID, item)`
ℓ`scan(namespace)` $\rightarrow$ `iterator`
`newData(namespace)` $\rightarrow$ `item`

Table 3: Provider API

It is important to note that locality in the key space does not guarantee locality in the network, although some proposed algorithms do try to minimize network distance for nearby keys. We return to this issue in Section 7.

3.2.2 Storage Manager

The storage manager is responsible for the temporary storage of DHT-based data while the node is connected to the network. The API for this layer is shown in Table 2. The API is designed to be easily realized via standard main-memory data structures, a disk-based indexing package like Berkeley DB [22], or simply a filesystem. All we expect of the storage manager is to provide performance that is reasonably efficient relative to network bottlenecks. The data in the DHT is distributed among many machines, and in many applications each machine will store a relatively small amount of queryable information. (This is even true for filesharing, for example, with respect to the index of filenames that gets queried). For simplicity in our early work, we use a main-memory storage manager. The modularity of the overall design allows for more complex and scalable storage managers to be used as required.

3.2.3 Provider

The provider is responsible for tying the routing layer and storage manager together while providing a useful interface to applications. The complete API is shown in Table 3.

Before describing the API, some notes on our DHT naming scheme are appropriate. Each object in the DHT has a namespace, resourceID, and instanceID. The namespace and resourceID are used to calculate the DHT key, via a hash function. The namespace 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, they are created implicitly when the first item is `put` and destroyed when the last item expires (as described below). The resourceID is generally intended to be a value that carries some semantic meaning about the object. Our query processor by default assigns the resourceID to be the value of the primary key for base tuples, although any attribute (or combination) could be used for this purpose. Items with the same namespace and resourceID will have the same key and thus map to the same node. 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 `put` and `get` calls are based directly on this naming scheme. Note that – as with most indexes – `get` is key-based, not instance-based, and therefore may return multiple items.

In order to adhere to our principle of "relaxed consistency", PIER uses the common Internet approach of *soft state* to achieve reliability and limit overheads in the face of frequent failures or disconnections [8]. This is the purpose of the "lifetime" argument of the `put` call – it gives the DHT a bound on how long it should store the item after receipt. Each producer of data can also periodically invoke the `renew` call to keep their information live for as long as they like. If a data item is not refreshed within the lifetime, then the item is deleted from the DHT on the responsible node. Thus, when a node fails or is disconnected, DHT entries are lost. However, these entries will be restored to the system as soon as the node re-sends that information when renewing.

To run a query, PIER attempts to contact the nodes that hold data in a particular namespace. A `multicast` communication primitive is used by the provider for this purpose [18]. The provider supports scan access to all the data stored locally on the node through the ℓ`scan` iterator. When run in parallel on all nodes serving a particular namespace, this serves the purpose of scanning a relation. Finally the provider supports the `newData` callback to the application to inform it when a new data item has arrived in a particular namespace.

3.3 QP Overview

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.

Unlike many traditional databases, we do not employ an iterator model to link operators together [13]. Instead operators produce results as quickly as possible (push) and enqueue the data for the next operator (pull). This intermediate queue is capable of hiding much of the network latency when data must be moved to another site. Since networking is such a fundamental aspect of our design, we chose not to encapsulate it away as in Volcano [12].

We intend in the future to add additional functionality to PIER's query processor, including system catalogs, an extensible operator interface, and declarative query parsing and optimization. Note that such additional modules are complementary to the query processor itself: a parser and optimizer will be layered above the existing query processor[4], and the catalog facility will reuse the DHT and query processor.

Following our "natural habitat" design principle, we do not currently provide facilities within PIER for modifying data managed by wrappers. Currently, we expect wrappers to insert, update, and delete items (or references to items) and tables (namespaces) directly via the DHT interface. Once we add a query parser, it would be fairly simple to provide DDL facilities for PIER to drive these DHT-based data modifications. But even if we did that, the updates would go to the soft state in the DHT, *not* to the wrappers. If the need to provide data-modification callbacks to updatable wrappers becomes important, such facilities could be added.

3.3.1 Data Semantics for PIER Queries

Given our "relaxed consistency" design principle, we provide a best effort data semantics in PIER that we call a *dilated-reachable snapshot*. First, we define a "reachable snapshot" as the set of data published by reachable nodes at the time the query is sent from the client node. As a practical matter, we are forced to relax reachable snapshot semantics to accommodate the difficulty of global synchronization of clocks and query processing. Instead, we define correct behavior in PIER based on the *arrival* of the query multicast message at reachable nodes: our reference "correct" data set is thus the (slightly time-dilated) 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. For the applications we are considering, this pragmatic asynchrony in snapshots seems acceptable. Of course, our actual query answers may not even provide this level of consistency, because of failures and partitions (published data from a reachable node may be transiently indexed by the DHT at a now-unreachable node), and soft state (the DHT may transiently index data at a reachable node that was published by a now-unreachable node).

4 DHT-Based Distributed Joins

Our join algorithms are adaptations of textbook parallel and distributed schemes, which leverage DHTs whenever possible. This is done both for the software elegance afforded by reuse, and because DHTs provide the underlying Internet-level scalability we desire. We use 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. In database terms, DHTs can serve as "exchange" mechanisms [12], as hash indexes, and as the hash tables that underlie many parallel join algorithms. DHTs provide these features in the face of a volatile set of participating nodes, a critical feature not available in earlier database work. As we will see below, we also use DHTs to route messages other than tuples, including Bloom Filters.

We have implemented two different binary equi-join algorithms, and two bandwidth-reducing rewrite schemes. We discuss these with respect to relations R and S. We assume that the tuples for R and S are stored in the DHT in separate namespaces N_R and N_S. We note here that PIER also provides a DHT-based temporary table facility for materializing operator output within query plans.

4.1 Core Join Algorithms

Our most general-purpose equi-join algorithm is a DHT-based version of the pipelining *symmetric hash join* [35], in-

[4]This would not be the case if we choose a continuously adaptive scheme like [1], which operates within a query plan. We discuss this further in Section 7.

terleaving building and probing of hash tables on each input relation. In the DHT context, all data is already hashed by some resourceID, so we speak of *rehashing* a table on the join key. To begin rehashing, each node in N_R or N_S performs an ℓscan to locate each R and S tuple. Each tuple that satisfies all the local selection predicates is copied (with only the relevant columns remaining) and must be put into a new unique DHT namespace, N_Q. The values for the join attributes are concatenated to form the resourceID for the copy, and all copies are tagged with their source table name.

Probing of hash tables is a local operation that occurs at the nodes in N_Q, in parallel with building. Each node registers with the DHT to receive a newData callback whenever new data is inserted into the local N_Q partition. When a tuple arrives, a get to the N_Q is issued to find matches in the other table; this get is expected to stay local. (If the local DHT key space has been remapped in the interim, the get will return the correct matches at the expense of an additional round trip.). Matches are concatenated to the probe tuple to generate output tuples, which are sent to the next stage in the query (another DHT namespace) or, if they are output tuples, to the initiating site of the query.

The second join algorithm, *Fetch Matches*, is a variant of a traditional distributed join algorithm that works when one of the tables, say S, is already hashed on the join attributes. In this case, N_R is ℓscanned, and for each R tuple a get is issued for the corresponding S tuple. Note that local selections on S do not improve performance – they do not avoid gets for each tuple of R, and since these gets are done at the DHT layer, PIER's query processor does not have the opportunity to filter the S tuples at the remote site (recall Figure 1). In short, selections on non-DHT attributes cannot be pushed into the DHT. This is a potential avenue for future streamlining, but such improvements would come at the expense of "dirtying" DHT APIs with PIER-specific features – a design approach we have tried to avoid in our initial implementation. Once the S tuples arrive at the corresponding R tuple's site, predicates are applied, the concatenation is performed, and results are passed along as above.

4.2 Join Rewriting

Symmetric hash join requires rehashing both tables, and hence can consume a great deal of bandwidth. To alleviate this when possible, we also implemented DHT-based versions of two traditional distributed query rewrite strategies, to try and lower the bandwidth of the symmetric hash join. Our first is a *symmetric semi-join*. In this scheme, we minimize initial communication by locally projecting both R and S to their resourceIDs and join keys, and performing a symmetric hash join on the two projections. The resulting tuples are pipelined into Fetch Matches joins on each of the tables' resourceIDs. (In our implementation, we actually issue the two joins' fetches in parallel since we know both fetches will succeed, and we concatenate the results to generate the appropriate number of duplicates.)

Our other rewrite strategy uses Bloom joins. First, Bloom Filters are created by each node for each of its local R and S fragments, and are published into a small temporary DHT namespace for each table. At the sites in the Bloom namespaces, the filters are OR-ed together and then multicast to all nodes storing the opposite table. Following the receipt of a Bloom Filter, a node begins ℓscanning its corresponding table fragment, but rehashing only those tuples that match the Bloom Filter.

5 Validation and Performance Evaluation

In this section we use analysis, simulations and experiments over a real network to demonstrate PIER.

Traditionally, database scalability is measured in terms of database sizes. In the Internet context, it is also important to take into account the network characteristics and the number of nodes in the system. Even when there are plenty of computation and storage resources, the performance of a system can degrade due to network latency overheads and limited network capacity. Also, although adding more nodes to the system increases the available resources, it can also increase latencies. The increase in latency is an artifact of the DHT scheme we use to route data in PIER (as described in Section 3.1). In particular, with CAN – the DHT scheme we use in our system – a data item that is sent between two arbitrary nodes in the system will traverse $n^{\frac{1}{4}}$ intermediate nodes on average (though recall that this increase could be reduced to logarithmic through changing the DHT parameters).

To illustrate these impacts on our system's performance we use a variety of metrics, including the maximum inbound traffic at a node, the aggregate traffic in the system, and the time to receive the last or the k-th result tuple. Finally, since the system answers queries during partial failures, we need to quantify the effects of these failures on query results.

We next present the load and the network characteristics we use to evaluate PIER.

5.1 Workload

In all our tests we use the following simple query as the workload:

```
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 R and S are synthetically generated. Unless otherwise specified, 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%. In addition, the last predicate uses a function f(x,y); since it references both R and S, any query plan must evaluate it after the equi-join. We chose the distribution of the join columns such that 90% of R tuples have one matching join tuple in S (before any other predicates are evaluated) and the remaining 10% have no matching tuple in S. The R.pad attribute is used to ensure that all result tuples are 1 KB in size. Unless otherwise specified, we use the symmetric hash-join strategy to implement the join operation.

5.2 Simulation and Experimental Setup

The simulator and the implementation use the same code base. The simulator allows us to scale up to 10,000 nodes, after which the simulation no longer fits in RAM – a limitation of simulation, not of the PIER architecture itself. The simulator's scalability comes at the expense of ignoring the cross-traffic in the network and the CPU and memory utilizations. We use two topologies in our simulations. The first is a fully connected network where the latency between any two nodes is 100 ms and the inbound link capacity of each node is 10 Mbps. This setup corresponds to a system consisting of homogeneous nodes spread throughout the Internet where the network congestion occurs at the last hop. In addition, we use the GT-ITM package [6] to generate a more realistic transit-stub network topology. However, since the fully-connected topology allows us to simulate larger systems, and since the simulation results on the two topologies are qualitatively similar (as we'll see in Section 5.7), we use the first topology in most of our simulations.

In addition, we make two simplifying assumptions. First, in our evaluation we focus on the bandwidth and latency bottlenecks, and ignore the computation and memory overheads of query processing. Second, we implicitly assume that data changes at a rate higher than the rate of incoming queries. As a result, the data needs to be shipped from source nodes to computation nodes for every query operation.

We also run experiments of PIER deployed (not simulated!) on the largest set of machines we had available to us – a shared cluster of 64 PCs connected by a 1-Gbps network.

All measurements reported in this section are performed after the CAN routing stabilizes, and tables R and S are loaded into the DHT.

5.3 Centralized vs. Distributed Query Processing

In standard database practice, centralized data warehouses are often preferred over traditional distributed databases. In this section we make a performance case for distributed query processing at the scales of interest to us. Consider the join operation presented in Section 5.1 and assume that tables R and S are distributed among n nodes, while the join is executed at m "computation" nodes, where $1 \leq m \leq n$.

If there are t bytes of data *in toto* that passed the selection predicates on R and S, then each of the computation nodes would need to receive $t/m - t/nm$ data on average. The second term accounts for the small portion of data that is likely to remain local. In our case the selectivity of the predicates on both R and S is 50%, which result in a value of t of approximately 1 GB for a database of 2 GB.

When there is only one computation node in a 2048-node network, one would need to provision for a very high link capacity in order to obtain good response times. For instance, even if we are willing to wait one minute for the results, one needs to reserve at least 137 Mbps for the downlink bandwidth, which would be very expensive in practice. We validated these calculation in our simulator, but do not present the data due to space limitations.

5.4 Scalability

One of the most important properties of any distributed system is the ability to scale its performance as the number of nodes increases. In this section, we evaluate the scalability of our system by proportionally increasing the load with the number of nodes.

Consider the query above, where each node is responsible for 1 MB of source data. Figure 3 plots the response time for the 30-th tuple. The value 30 was chosen to be a bit after the first tuple received, and well before the last. We avoid using the first response as a metric here, on the off chance that it is produced locally and does not reflect network limitations. In this scalability experiment we are not interested in the time to receive the last result, because as we increase the load and network size, we increase the number of results; at some point in that exercise we end up simply measuring the (constant) network capacity of the query site, where all results must arrive.

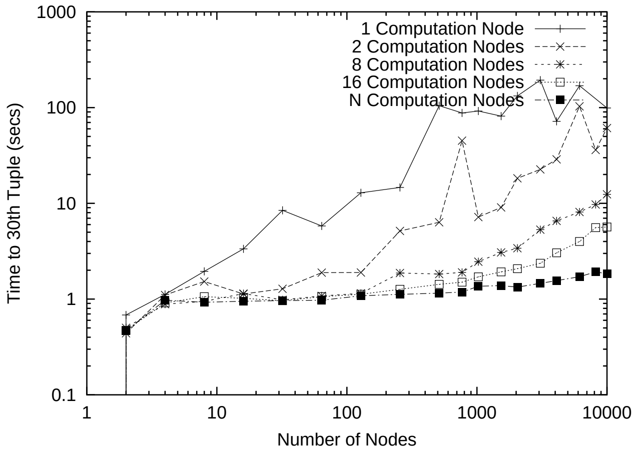

Figure 3: Average time to receive the 30-th result tuple when both the size of the network and the load are scaled up. Each data point is averaged over three independent simulations.

As shown in Figure 3, when all nodes participate in the computation the performance of the system degrades only by a factor of 4 when the network size and the load increase from two to $10,000$ nodes. We are unable to obtain perfect scaling because the number of overlay hops for each lookup increases as the network size increases. This ultimately leads to an increase in the lookup latency. In particular, with our implementation of CAN the lookup length increases with $n^{1/4}$, where n is the number of nodes in the system. Thus, as the number of nodes increases from two to $10,000$, the lookup length increases by a factor of about 10. The reason we observe only a factor of 4 degradation in our measurements is that, besides the lookup operation, there is a fixed overhead associated with the join operation[5]. We discuss these overheads in more detail in Section 5.5. Finally, note that the increase in the lookup length could be reduced by choosing a different value for d in CAN or using a different DHT design.

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

5.5 Join Algorithms and Rewrite Strategies

In this section we evaluate the four join strategies described in Section 6: symmetric hash join, Fetch Matches, symmetric semi-join rewriting, and Bloom Filter rewriting. We consider two simulation scenarios where the bottleneck is the latency or the network capacity. Note that the former case is equivalent to a system in which the network capacity is infinite.

[5]For example, the time it takes to send the result tuple back to the join initiator doesn't change as the network size increases.

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

Table 4: Average time to receive the last result tuple.

5.5.1 Infinite Bandwidth

To quantify the impact of the propagation delay on these four strategies, we first ignore the bandwidth limitations, and consider only the propagation delay.

Each strategy requires distributing the query instructions to all nodes (a multicast message) and the delivery of the results (direct IP communication between nodes). Table 4 shows the average time measured by the simulator to receive the last result tuple by the query node in a network with $n = m = 1024$ nodes. We proceed to explain these values analytically.

Recall that the lookup in CAN takes $n^{\frac{1}{4}}$ hops on average. Since the latency of each hop is 100 ms, the average lookup latency is $n^{\frac{1}{4}} \times 100 = 0.57$ sec. In contrast, the latency of a direct communication between any two nodes is 100 ms. Reference [18] describes the multicast operation, used to distribute the query processing, in detail. Here we only note that in this particular case it takes the multicast roughly 3 sec to reach all nodes in the system. Next, we detail our analysis for each join strategy:

Symmetric hash join To rehash the tuples, the DHT must (1) lookup the node responsible for a key, and (2) send the `put` message directly[6] to that node. Adding the multicast and the latency for delivering the results to the join initiator, we obtain $3 + 0.57 + 2 * 0.1 = 3.77$ sec, which is close to the simulator's value in Table 4.

Fetch Matches. To find a possible matching tuple, the nodes must (1) lookup the address of the node responsible for that tuple, (2) send a request to that node, (3) wait for the reply, and (4) deliver the results. In this scenario there is one CAN lookup and three direct communications. Adding up the costs of all these operations yields $3 + 0.57 + 3 * 0.1 = 3.87$ sec.

Symmetric semi-join rewrite. In this case, the projected tuples are (1) inserted into the network (one CAN lookup plus one direct communication), and (2) a Fetch Matches join is performed over the indexes (one CAN lookup and one direct communication). Thus, we have a total of two CAN lookup operations and four direct communications (including the delivery of results). All these operations account for $3 + 2 * 0.57 + 4 * 0.1 = 4.54$ sec.

Bloom Filter rewrite. Each node creates the local Bloom Filters and sends them to the collectors (one lookup and one direct communication). In turn, the collectors distribute the filters back to the source nodes (one multicast), and the source nodes then perform the rehash for a symmetric hash join (one CAN lookup and one direct communication). Together, in addition to the multicast operation required to distribute the query processing, we have another multicast operation, two lookup operations, and three direct communications. Adding them up gives us $2*3+2*0.57+3*0.1 = 7.44$ sec. The reason that this value is larger than the one reported in Table 4 is because in our derivations we have assumed that there is one node that experiences worst case delays when waiting for both multicasts. However, this is very unlikely to happen in practice.

[6] Most DHT operations consist of a lookup followed by direct communication. Since these two operations are not atomic, there can be a case where a node continuously fails to contact a node responsible for a certain key, because the node mapping to that key always changes after the lookup is performed. However, this is unlikely to be a problem in practice. The bandwidth savings of not having a large message hop along the overlay network outweighs the small chance of this problem.

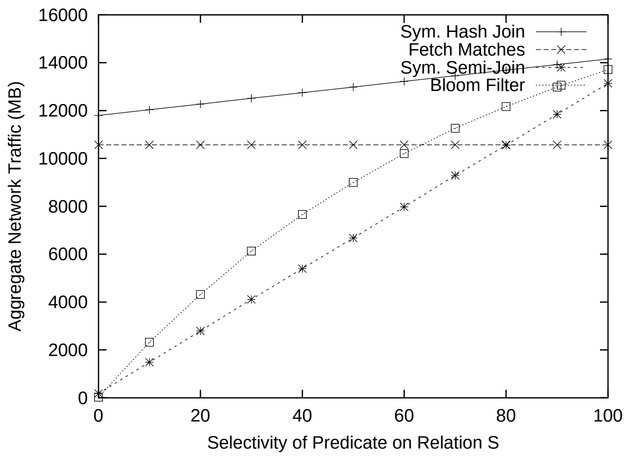

Figure 4: Aggregate network traffic generated by each join strategy.

5.5.2 Limited Bandwidth

In this section, we evaluate the performance of the four join strategies in the baseline simulation setup in which the inbound capacity of each node is 10 Mbps. We first measure the network overhead incurred by each join strategy, and then measure the time to receive the last result tuple.

Figure 4 plots the bandwidth requirements for each strategy as a function of the selectivity of the predicate on S. The total size of relations R and S is approximately 25 GB, and the system has 1024 nodes.

As expected, the symmetric hash join uses the most network resources since both tables are rehashed. The increase in the total inbound traffic is due to the fact that both the number of tuples of S that are rehashed and the number of results increase with the selectivity of the selection on S. In contrast, the 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. Finally, 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.

To evaluate the performance of the four algorithms, in Figure 5 we plot the average time to receive the last result tuple.

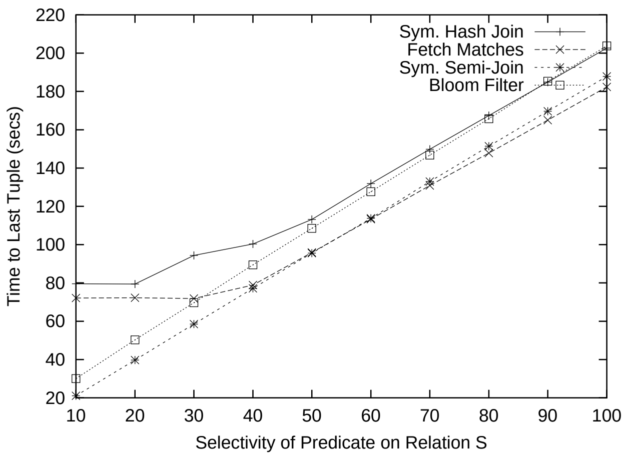

Figure 5: Time to receive the last result tuple for each strategy.

The reason we use the last tuple here rather than the 30-th is to illustrate the different bottlenecks in the system. When the selectivity of the predicate on S is lower than 40%, the bottleneck is the inbound capacity of the computation nodes. As a result, the plots in Figure 5 follow a trend similar to that of the total inbound traffic shown in Figure 4. As the predicate selectivity on S exceeds 40% the number of results increases enough such that the bottleneck switches to being the inbound capacity of the query site.

5.6 Effects of Soft State

In this section we evaluate the robustness of our system in the face of node failures. The typical algorithm used by DHTs to detect node failures is for each node to send periodic keep-alive messages to its neighbors. If a certain number of keep-alive messages remain unanswered, a node will conclude that its neighbor has failed. Thus, when a node fails, it will take its neighbors some time until they learn that the node has failed. During this time all the packets sent to the failed node are simply dropped. In this section we assume somewhat arbitrarily that it takes 15 seconds to detect a node failure. Once a node detects a neighbor failure, we assume that the node will route immediately around the failure.

When a node fails, all the tuples stored at that node are lost – even if the nodes that published them are still reachable. A simple scheme to counteract this problem is to periodically `renew` (refresh) all tuples. To evaluate this scheme we plot the average recall as a function of the node failure rate for different refresh periods when there are 4096 nodes in the system (see Figure 6). A refresh period of 60 sec means that each tuple is refreshed every 60 sec. Thus, when a node fails, the tuples stored at that node are unavailable for 30 sec on average. As expected the average recall decreases as the failure rate increases, and increases as the refresh period decreases. For illustration, consider the case when the refresh period is 60 sec and the failure rate is 240 nodes per minute. This means that about 6% (i.e., $240/4096$) of the nodes fail every minute. Since it takes up to 30 sec on average until a lost tuple is reinserted in the system, we expect that $6 * 30\text{sec}/60\text{sec}$ = 3% of the live tuples in the system to be unavailable. This would result in a recall of 97% which is close to the value of 96% ploted in Figure 6. Note that this recall figure is with

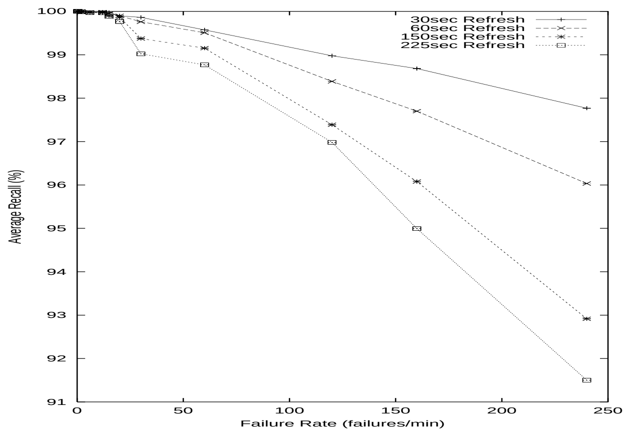

Figure 6: Average recall for different refresh periods.

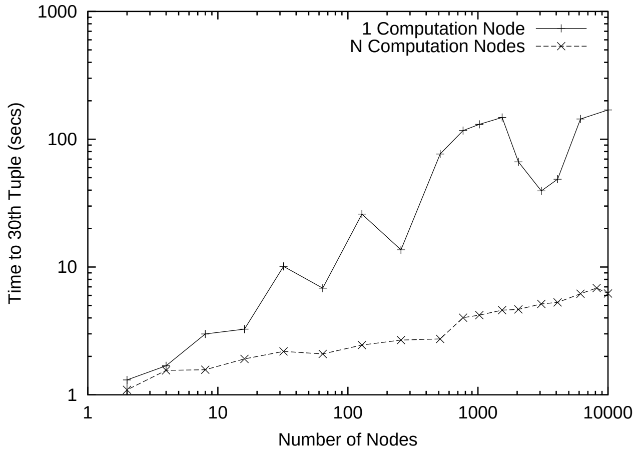

Figure 7: Average time to receive the 30-th result tuple when both the size of the network and the load are scaled up for a transit stub topology (compare this with the plot in Figure 3). Each data point is averaged over three simulations.

respect to the reachable snapshot semantics presented in Section 3.3.

5.7 Transit Stub Topology

So far, in our simulations we have used a fully-connected network topology. A natural question is whether using a more complex and realistic network topology would change the results. In this section, we try to answer this question by using the GT-ITM package [6] to generate a transit stub network topology. The network consists of four transit domains. There are 10 nodes per transit domain, and there are three stub domains per transit node. The number of nodes in the system are distributed uniformly among the stub domains. The transit-to-transit latency is 50 ms, the transit-to-stub latency is 10 ms, and the latency between two nodes within the same stub is 2 ms. The inbound link to each node is 10 Mbps.

Figure 7 shows the results of reruning the scalability experiments from Section 5.4 using the transit-stub topology. The results exhibit the same trends as the results obtained by using the fully-connected topology (see Figure 3). The only

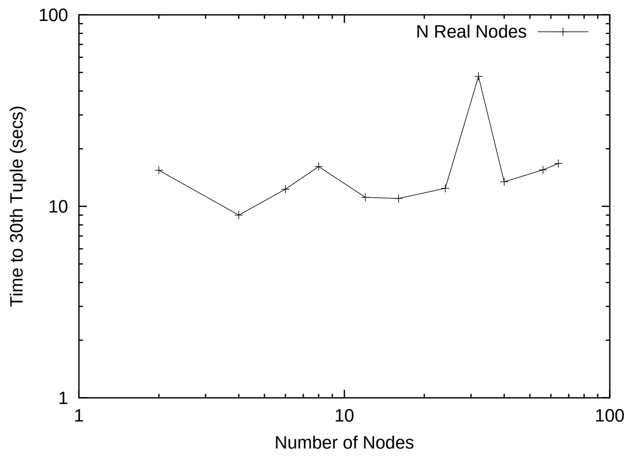

Figure 8: Average time to receive the 30-th result tuple in our prototype implementation. All nodes in the system are also computation nodes. Each data point is averaged over seven independent runs.

significant difference is that the absolute values to receive the 30-th result tuple are larger. This is because the average end-to-end delay between two nodes in the transit stub topology is about 170 ms, instead of 100 ms as in the case of the fully connected graph. Also note that in Figure 7 we plot the results only for up 4096 nodes instead of 10,000 nodes. This was the maximum network size allowed by our current simulator using the transit-stub topology. This limitation, together with the fact that we did not observe any qualitative different results for the two topologies, were the main reasons for using the fully-connected topology in this paper.

5.8 Experimental Results

In this section we present the experimental results of running our prototype implementation on a cluster of 64 PCs connected by an 1 Gbps network. Figure 8 plots the time to receive the 30-th result tuple as the number of nodes increases from 2 to 64 and the load scales accordingly. 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. We believe the peak in response time at 32 nodes is due to an artifact in our CAN implementation.

6 Related Work

Our work on PIER was inspired and informed by a number of research traditions. We attempt to provide a rough overview of related work here.

6.1 Widely-Deployed Distributed Systems

The leading example of a massively distributed system is the Internet itself. The soft-state consistency of the Internet's internal data [8] is one of the chief models for our work. On the schema standardization front, we note that significant effort is expended in standardizing protocols (e.g. IP, TCP, SMTP, HTTP) to ensure that the "schema" of messages is globally agreed-upon, but that these standards are often driven by popularly deployed software. While rarely stored persistently, the number of bytes generated from each of these "schemas" annually is enormous.

The most prevalent distributed query systems are P2P file-sharing and DNS [21]. Both are examples of globally standardized schemas, and both make significant sacrifices in data consistency in order to scale: neither provides anything like transactional guarantees. Filesharing systems today do not necessarily provide full recall of all relevant results, and often provide poor precision by returning docIDs that are currently inaccessible. DNS does a better job on recall, but also keeps stale data for a period of time and hence can sacrifice precision. The scalability and adoption model of these systems is another model for our work here.

6.2 Database Systems

Query processing in traditional distributed databases focused on developing bandwidth-reduction schemes, including semi-joins and Bloom joins, and incorporated these techniques into traditional frameworks for query optimization [23]. Mariposa was perhaps the most ambitious attempt at geographic scaling in query processing, attempting to scale to thousands of sites [32]. Mariposa focused on overcoming cross-administrative barriers by employing economic feedback mechanisms in the cost estimation of a query optimizer. To our knowledge, Mariposa was never deployed or simulated on more than a dozen machines, and offered no new techniques for query *execution*, only for query optimization and storage replication. By contrast, we postpone work on query optimization in our geographic scalability agenda, preferring to first design and validate the scalability of our query execution infrastructure.

Many of our techniques here are adaptations of query execution strategies used in parallel database systems [10]. Unlike distributed databases, parallel databases have had significant technical and commercial impact. While parallelism *per se* is not an explicit motivation of our work, algorithms for parallel query processing form one natural starting point for systems processing queries on multiple machines.

6.3 P2P Database and IR Proposals

P2P databases are a growing area of investigation. An early workshop paper focused on storage issues [14], which we intentionally sidestep here – our design principles for scalability lead us to only consider soft-state storage in PIER. A related body of work is investigating the semantic data integration challenges in autonomous P2P databases (e.g. [15, 3].) Solutions to those problems are not a prerequisite for the impact of our work, but would nicely complement it.

This paper builds on our initial workshop proposal for PIER [16]. To our knowledge, this paper presents the first serious treatment of scalability issues in a P2P-style relational query engine. There is an emerging set of P2P text search proposals [27, 33] that are intended to provide traditional IR functionality. These are analogous to a workload-specific relational query engine, focusing on Bloom-Filter-based intersections of inverted index posting lists.

6.4 Network Monitoring

A number of systems have been proposed for distributed network monitoring. The closest proposal to our discussion here is Astrolabe, an SQL-like query system focused specifically on aggregation queries for network monitoring [34]. Astrolabe provides the ability to define materialized aggregation views over sub-nets, and to run queries that hierarchically compose these views into coarser aggregates. Astrolabe provides a constrained subset of SQL that sacrifices general query facilities in favor of a family of queries that exploit this hierarchy efficiently. This contrasts with the flat topology and general platform provided by PIER.

Another workshop proposal for peer-to-peer network monitoring software is presented in [30], including a simple query architecture, and some ideas on trust and verification of measurement reports.

6.5 Continuous Queries and Streams

A final related body of work is the recent flurry of activity on processing continuous queries over data streams; these proposals often use network monitoring as a driving application [2]. Certainly continuous queries are natural for network monitoring, and this body of work may be especially relevant here in its focus on data compression ("synopses") and adaptive query optimization. To date, work on querying streams has targeted centralized systems.

Somewhat more tangential are proposals for query processing in wireless sensor networks [5, 20]. These systems share our focus on peer-to-peer architectures and minimizing network costs, but typically focus on different issues of power management, extremely low bandwidths, and very lossy communication channels.

7 Conclusions and Future Work

In this paper we present the initial design and implementation of PIER, a structured query system intended to run at Internet scale. PIER is targeted at *in situ* querying of data that preexists in the wide area. To our knowledge, our demonstration of the scalability of PIER to over 10,000 nodes is unique in the database literature, even on simulated networks. Our experiments on actual hardware have so far been limited by the machines available to us, but give us no reason to doubt the scalability shown in our simulation results. We are currently deploying PIER on the PlanetLab testbed [25], which will afford us experience with a large collection of nodes distributed across the Internet.

The scalability of PIER derives from a small set of relaxed design principles, which led to some of our key decisions, including: the adoption of soft state and dilated-reachable snapshot semantics; our use of DHTs as a core scalability mechanism for indexing, routing and query state management; our use of recall as a quality metric; and our applications in network monitoring.

In this initial work we focused on the query execution aspects of PIER, and we believe this initial design thrust was sound. Our scalability results for "hand-wired" queries encourage us to pursue a number of additional research thrusts. These include the following:

Network Monitoring Applications: The existing PIER implementation is nearly sufficient to support some simple but very useful network monitoring applications, a topic of particular interest to the networking researchers among us. Implementing a handful of such applications should help us to prioritize our work on the many system design topics we discuss next.

Recursive Queries on Network Graphs: Computer networks form complex graphs, and it is quite natural to recursively query them for graph properties. As a simple example, in the Gnutella filesharing network it is useful to compute the set of nodes reachable within k hops of each node. A twist here is that *the data is the network*: the graph being queried is in fact the communication network used in execution. This very practical recursive query setting presents interesting new challenges in efficiency and robustness.

Hierarchical aggregation and DHTs. In this paper we focused on implementation and analysis of distributed joins. We have also implemented DHT-based hash grouping and aggregation in PIER in a straightforward fashion, analogous to what is done in parallel databases. However, parallel databases are designed for bus-like network topologies, and the techniques for aggregation are not necessarily appropriate in a multi-hop overlay network. Other in-network aggregation schemes, like those of Astrolabe [34] and TAG [20], perform *hierarchical* aggregation in the network, providing reduced bandwidth utilization and better load-balancing as a result. It is not clear how to leverage these ideas in a DHT-based system like PIER. One possible direction is to leverage the application callbacks supported during intermediate routing hops in many DHTs – data could be aggregated as it is routed, somewhat like the scheme in TAG. However it is not clear how to do this effectively. An alternative is to superimpose an explicit hierarchy on the DHT, but this undercuts the basic DHT approach to robustness and scalability.

Efficient range predicates: Because DHTs are a hashing mechanism, we have focused up to now on equality predicates, and in particular on equi-joins. In future, it is important for PIER to efficiently support other predicates. Foremost among these are the standard unidimensional range predicates typically supported in database systems by B-trees. Other important predicates include regular expressions and other string-matching predicates, multidimensional ranges, and near-neighbor queries.

Catalogs and Query Optimization: We have seen that our existing boxes-and-arrows-based query engine scales, but for usability and robustness purposes we would prefer to support declarative queries. This necessitates the design of a catalog manager and a query optimizer. A catalog is typically small, but has more stringent availability and consistency requirements than most of the data we have discussed up to now, which will stress some of our design principles. On the query optimization front, one approach is to start from classic parallel [17] and distributed database approaches [19], and simply enhance their cost models to reflect the properties of DHTs. This may not work well given the heterogeneity and shifting workloads on the wide-area Internet. We are considering mid-query adaptive optimization approaches like eddies [1] to capture changes in performance; adaptivity is especially attractive if we focus on continuous queries as we discuss below.

Continuous queries over streams: As noted in Section 6, we concur with prior assertions that continuous queries are natural over network traces, which may be wrapped as unbounded data streams. PIER already provides a pipelining query engine with an asynchronous threading model, so we can already process queries over wrapped distributed "streams" by introducing "windowing" schemes into our join and aggregation code. Beyond this first step, it would be interesting to see how proposed stream techniques for synopses, adaptivity, and sharing [2] could be adapted to our massively distributed environment.

Routing, Storage, and Layering: A number of potential optimizations center on PIER's DHT layer. These include: more efficient routing schemes to provide better physical locality in the network; "pushdown" of selections into the DHT, batch-routing of many tuples per call via the DHT; caching and replication of DHT-based data; and load-balancing of the DHT, especially in the face of heterogeneous nodes and links. Many of these are topics of active research in the growing DHT design community. An interesting question to watch is whether the DHT community efforts will be useful to the needs of our query processing system, or whether we will do better by designing query-specific techniques. In the latter case, a subsidiary question will be whether our unique needs can be addressed above the DHT layer, or whether we need specialized DHT support for increased performance.

8 Acknowledgments

This research was funded by NSF grants ANI-0133811, ANI-0196514, ANI-0207399, ANI-0225660, EIA-0122599, IIS-0205647, IIS-0209108, ITR-0081698, ITR-0121555, and ITR-0205519.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, Dallas, May 2000.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Madison, June 2002. ACM.

[3] P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing : A vision. In *Fifth International Workshop on the Web and Databases (WebDB 2002)*, June 2002.

[4] P. A. Bernstein, M. L. Brodie, S. Ceri, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, J. Gray, G. Held, J. M. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. F. Naughton, H. Pirahesh, M. Stonebraker, and J. D. Ullman. The asilomar report on database research. *SIGMOD Record*, 27(4):74–80, 1998.

[5] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proc. Mobile Data Management*, volume 1987 of *Lecture Notes in Computer Science*, Hong Kong, Jan. 2001. Springer.

[6] K. Calvert and E. Zegura. GT internetwork topology models (GT-ITM). `http://www.cc.gatech.edu/projects/gtitm/gt-itm/README`.

[7] J. Claessens, B. Preneel, and J. Vandewalle. Solutions for anonymous communication on the Internet. In *IEEE ICCST*, 1999.

[8] D. D. Clark. The design philosophy of the DARPA internet protocols. In *Proceedings SIGCOMM '88*, Aug. 1988.

[9] I. S. Consortium. Network wizards internet domain survey. `http://www.isc.org/ds/host-count-history.html`.

[10] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *CACM*, 35(6):85–98, 1992.

[11] 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.

[12] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. 1990 ACM SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, May 1990. ACM Press.

[13] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[14] S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What can database do for peer-to-peer? In *Proc. Fourth International Workshop on the Web and Databases (WebDB 2001)*, Santa Barbara, May 2001.

[15] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *19th International Conference on Data Engineering*, Bangalore, India, 2003.

[16] 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.

[17] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford University, 1995.

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

[19] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *Proc. Twelfth International Conference on Very Large Data Bases (VLDB '86)*, pages 149–159, Kyoto, Aug. 1986.

[20] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Dec. 2002.

[21] P. Mockapetris. Domain names – implementation and specification, Nov. 1987.

[22] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *Proc. 1999 Summer Usenix Technical Conference*, Monterey, June 1999.

[23] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999.

[24] J. Padhye and S. Floyd. Identifying the TCP behavior of web servers. In *Proceedings SIGCOMM '01*, June 2001.

[25] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. ACM HotNets-I Workshop*, Princeton, Oct. 2002.

[26] 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.

[27] P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. `http://issg.cs.duke.edu/search/`, June 2002.

[28] M. Roesch. Snort – lightweight intrusion detection for networks. In *13th USENIX Systems Administration Conference (LISA '99)*, Seattle, WA, Nov. 1999.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[30] S. Srinivasan and E. Zegura. Network measurement as a cooperative enterprise. In *Proc. First International Workshop on Peer-to-Peer Systems (IPTPS '01)*, Cambridge, MA, Mar. 2002.

[31] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: Scalable Peer-To-Peer lookup service for internet applications. In *Proc. 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[32] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[33] C. Tang, Z. Xu, and M. Mahalingam. psearch: Information retrieval in structured overlays. In *HotNets-I*, October 2002.

[34] R. van Renesse, K. P. Birman, D. Dumitriu, and W. Vogel. Scalable management and data mining using astrolabe. In *Proc. First International Workshop on Peer-to-Peer Systems (IPTPS '01)*, Cambridge, MA, Mar. 2002.

[35] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Info. Sys. (PDIS)*, pages 68–77, 1991.

[36] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.