The Architecture of PIER: an Internet-Scale Query Processor

Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis,
Timothy Roscoe, Scott Shenker, Ion Stoica and Aydan R. Yumerefendi

UC Berkeley and Intel Research Berkeley
p2p@db.cs.berkeley.edu

1 Introduction

This paper presents the architecture of PIER[1], an Internet-scale query engine we have been building over the last three years. PIER is the first general-purpose relational query processor targeted at a peer-to-peer (p2p) architecture of thousands or millions of participating nodes on the Internet. It supports massively distributed, database-style dataflows for snapshot and continuous queries. It is intended to serve as a building block for a diverse set of Internet-scale information-centric applications, particularly those that tap into the standardized data readily available on networked machines, including packet headers, system logs, and file names.

In earlier papers we presented the vision for PIER, its application relevance, and initial simulation results [28, 32]. We have also presented real-world results showing the benefits of using PIER in a p2p filesharing network [41, 43]. In this paper we present, for the first time, a detailed look at PIER's architecture and implementation. Implemented in Java, PIER targets an unusual design point for a relational query engine, and its architecture reflects the challenges at all levels, from the core runtime system through its aggressive multi-purpose use of overlay networks, up into the implementation of query engine basics including data representation, query dissemination, query operators, and its approach to system metadata. In addition to reporting on PIER's architecture, we discuss additional design concerns that have arisen since the system has become real, which we are addressing in our current work.

1.1 Context

Distributed database systems have long been a topic of interest in the database research community. The fundamental goal has generally been to make distribution *transparent* to users and applications, encapsulating all the details of distribution behind standard query language semantics with ACID guarantees. The resulting designs, such as SDD-1 [5], R* [40] and Mariposa [64], differ in some respects, but they share modest targets for network scalability: none of these systems has been deployed on much more than a handful of distributed sites.

The Internet community has become interested in distributed query processing for reasons akin to those we laid out in earlier work [32]. Not surprisingly, they approach this problem from a very different angle than the traditional database literature. The fundamental goal of Internet systems is to operate at very large scale (thousands if not millions of nodes). Given the inherent conflicts between consistency, availability, and tolerance to network partitions (the CAP theorem [7]), designers of Internet systems are willing to tolerate loose consistency semantics in order to achieve availability. They also tend to sacrifice the flexibility of a SQL-style database query language in favor of systems that scale naturally on the hierarchical wiring of the Internet. Examples in this category include Astrolabe [66] and IrisNet [22], two hierarchical Internet query systems that we discuss in Section 5.

PIER, coming from a mixed heritage, tries to strike a compromise between the Internet and database approaches. Like Internet systems, PIER is targeted for very large scales and therefore settles for relaxed semantics. However, PIER provides a full degree of data independence, including a relational data model and a full suite of relational query operators and indexing facilities that can manipulate data without regard to its location on the network.

We begin (Section 2) by describing some of the basic design choices made in PIER, along with the characteristics of target applications. This is followed (Section 3) by a detailed explanation of the PIER architecture. We also highlight key design challenges that we are still exploring in the system. In Section 4, we outline our future work on two important fronts: security and query optimization. Related work is presented in Section 5 and we conclude in Section 6.

2 Design Decisions and Sample Application

Many of the philosophical assumptions we adopted in PIER were described in an earlier paper [32]; that discussion guided our architecture. Here we focus on concrete design decisions we made in architecting the system. We also overview a num-

[1] PIER stands for Peer-to-peer Information Exchange and Retrieval.

Proceedings of the 2005 CIDR Conference

ber of sample applications built using PIER.

2.1 Design Decisions

PIER fully embraces the notion of *data independence*, and extends the idea from its traditional disk-oriented setting to promising new territory in the volatile realm of Internet systems [29]. PIER adopts a relational data model in which data values are fundamentally independent of their physical location on the network. While this approach is well established in the database community, it is in stark contrast to other Internet-based query processing systems, including well-known systems like DNS [49] and LDAP [30], filesharing systems like Gnutella and KaZaA, and research systems like Astrolabe [66] and IrisNet [22] – all of which use hierarchical networking schemes to achieve scalability. Analogies to the early days of relational databases are apropos here. PIER may be somewhat less efficient than a customized locality-centric solution for certain constrained workloads. But PIER's data-independence allows it to achieve reasonable performance on a far wider set of queries, making it a good choice for easy development of new Internet-scale applications that query distributed information.

2.1.1 Network Scalability, Resilience and Performance

PIER achieves scalability by using *distributed hash table* (DHT) technology (see [57, 60, 63] for a few representative references). As we discuss in more detail in Section 3.2, DHTs are overlay networks providing both location-independent naming and network routing, and they are reused for a host of purposes in PIER that are typically separate modules in a traditional DBMS (Section 3.3.6). DHTs are extremely scalable, typically incurring per-operation overheads that grow only logarithmically with the number of machines in the system. They are also designed for resilience, capable of operating in the presence of *churn* in the network: frequent node and link failures, and the steady arrival and departure of participating machines in the network.

PIER is designed for the Internet, and assumes that the network is the key bottleneck. This is especially important for a p2p environment where most of the hosts see bottlenecks at the "last mile" of DSL and cable links. As discussed in [32], PIER minimizes network bandwidth consumption via fairly traditional bandwidth-reducing algorithms (e.g., Bloom Joins [44], multi-phase aggregation techniques [62], etc). But at a lower and perhaps more fundamental system level, PIER's core design centers around the low-latency processing of large volumes of network messages. In some respects therefore it resembles a router as much as a database system.

2.1.2 Decoupled Storage

A key decision we made in our earliest discussions was to decouple storage from the query engine. We were inspired in this regard by p2p filesharing applications, which have been successful in adding new value by querying pre-existing data *in situ*. This approach is also becoming common in the database community in data integration and stream query processing systems. PIER is designed to work with a variety of storage systems, from transient storage and data streams (via main memory buffers) to locally reliable persistent storage (file systems, embedded DBs like BerkeleyDB, JDBC-enabled databases), to proposed Internet-scale massively distributed storage systems [15, 39]. Of course this decision also means that PIER sacrifices the ACID storage semantics of traditional distributed databases. We believe this is a natural trade-off in the Internet-scale context since (a) many Internet-scale applications do not need persistent storage, and (b) the CAP theorem suggests that strong consistency semantics are an unrealistic goal for Internet-scale systems.

In strictly decoupling storage from the query engine, we give up the ability to reliably store system metadata. As a result, PIER has *no metadata catalog* of the sort found in a traditional DBMS. This has significant ramifications on many parts of our system (Sections 3.3.1, 3.3.2, and 4.2).

2.1.3 Software Engineering

From day one, PIER has targeted a platform of many thousands of nodes on a wide-area network. Development and testing of such a massively distributed system is hard to do in the lab. In order to make this possible, *native simulation* is a key requirement of the system design. By "native" simulation we mean a runtime harness that emulates the network and multiple processors, but otherwise exercises the standard system code.

The trickiest challenges in debugging massively distributed systems involve the code that deals with distribution and parallelism, particularly the handling of node failures and the logic surrounding the ordering and timing of message arrivals. These issues tend to be very hard to reason about, and are also difficult to test robustly in simulation. As a result, we attempted to encapsulate the distribution and parallelism features within as few modules as possible. In PIER, this logic resides largely within the DHT code. The relational model helps here: while subtle network timing issues can affect the ordering of tuples in the dataflow, this has no effect on query answers or operator logic (PIER uses no distributed sort-based algorithms).

2.2 Potential Applications

PIER is targeted at applications that run on many thousands of end-users' nodes where centralization is undesirable or infeasible. To date, our work has been grounded in two specific application classes:

- **P2P File Sharing.** Because this application is in global deployment already, it serves as our baseline for scalability. It is characterized by a number of features: a simple schema (keywords and fileIDs), a constrained query workload (Boolean keyword search), data that is stored without any inherent locality, loose query semantics (resolved by users), relatively high churn, no administration, and extreme ease of use. In order to test PIER, we implemented a filesharing search engine using PIER and

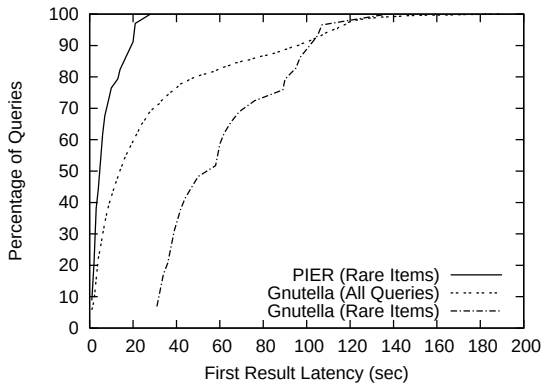

Figure 1: CDF of latency for receipt of an answer from PIER and Gnutella, over real user queries intercepted from the Gnutella network. PIER was measured on 50 PlanetLab nodes worldwide, over a challenging subset of the queries: those that used "rare" keywords used infrequently in the past. As a baseline, the CDF for Gnutella is presented both for the rare-query subset, and for the complete query workload (both popular and rare queries). Details appear in [41].

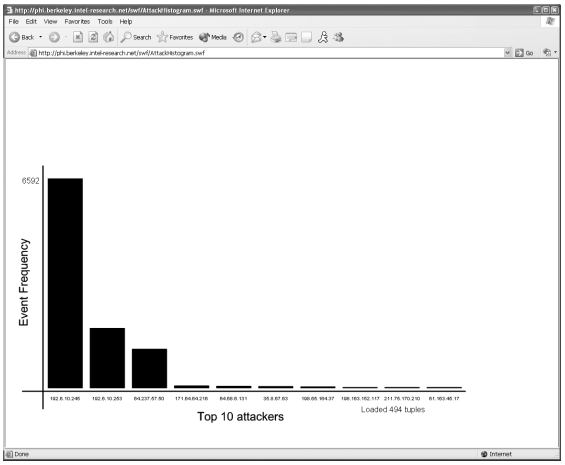

Figure 2: The top 10 sources of firewall log events as reported by 350 PlanetLab nodes running on 5 continents.

integrated it into the existing Gnutella filesharing network, to yield a hybrid search infrastructure that uses the Gnutella protocol to find widely replicated nearby items, and the PIER engine to find rare items across the global network. As we describe in a paper on the topic [41], we deployed this hybrid infrastructure on 50 nodes worldwide in the PlanetLab testbed [54], and ran it over real Gnutella queries and data. Our hybrid infrastructure outperformed native Gnutella in both recall and performance. As one example of the results from that work, the PIER-based hybrid system reduced the number of Gnutella queries that receive no results by 18%, with significantly lower answer latency. Figure 1 presents a representative performance graph from that study showing significant decreases in latency.

- **Endpoint Network Monitoring.** End-hosts have a wealth of network data with standard schemas: packet traces and firewall logs are two examples. Endpoint network monitoring over this data is an important emerging application space, with a constrained query workload (typically distributed aggregation queries with few if any joins), streaming data located at both sources and destinations of traffic, and relatively high churn. Approximate answers and online aggregation are desirable.

  Figure 2 shows a prototype applet we built, which executes a PIER query running over firewall logs on 350 PlanetLab nodes worldwide. The query reports the IP addresses of the top ten sources of firewall events across all nodes. Recent forensic studies of (warehoused) firewall logs suggest that the top few sources of firewall events generate a large fraction of total unwanted traffic [74]. This PIER query illustrates the same result in real time, and could be used to automatically parameterize packet filters in a firewall.

3 Architecture

In this section we describe the PIER architecture in detail. We begin with the low-level execution environment and overview of the DHT. We then discuss the "life of a query", present details of the query processing logic and highlight the varying ways the DHT is used in query processing.

3.1 Execution Environment

Like any serious query engine, PIER is designed to achieve a high degree of multiprogramming across many I/O-bound activities. As noted in Section 2, it also needs to support native simulation. These requirements led us to a design grounded in two main components: a narrow *Virtual Runtime Interface*, and an *event-based* style of multiprogramming that makes minimal use of threads.

3.1.1 Virtual Runtime Interface

The lowest level of PIER presents a simple *Virtual Runtime Interface* (VRI) that encapsulates the basic execution platform. The VRI can be bound to either the real-world Physical Runtime Environment (Section 3.1.3) or to a Simulation Environment (Section 3.1.4). The VRI is composed of interfaces to the clock and timers, to network protocols, and to the internal PIER scheduler that dispatches clock and network events. For the interested reader, a representative set of the methods provided by the VRI are shown in Table 1.

3.1.2 Events and Handlers

Multiprogramming in PIER is achieved via an event-based programming model running in a single thread. This is common in routers and network-bound applications, where most computation is triggered by the arrival of a message, or by tasks that are specifically posted by local code. All events in PIER are processed by a single thread with no preemption.

Clock and Main Scheduler

long $getCurrentTime$()
void $scheduleEvent$(delay, callbackData, callbackClient)
void $handleTimer$(callbackData)

UDP

void $listen$(port, callbackClient)
void $release$(port)
void $send$(source, destination, payload, callbackData, callbackClient)
void $handleUDPAck$(callbackData, success)
void $handleUDP$(source, payload)

TCP

void $listen$(port, callbackClient)
void $release$(port)
TCPConnection $connect$(source, destination, callbackClient)
$disconnect$(TCPConnection)
int $read$(byteArray)
int $write$(byteArray)
void $handleTCPData$(TCPConnection)
void $handleTCPNew$(TCPConnection)
void $handleTCPError$(TCPConnection)

Table 1: Selected methods in the VRI.

The single-threaded, event-based approach has a number of benefits for our purposes. Most importantly, it supports our goal of native simulation. Discrete-event simulation is the standard way to simulate multiple networked machines on a single node [53]. By adopting an event-based model at the core of our system, we are able to opaquely reuse most of the program logic whether in the Simulation Environment or in the Physical Runtime Environment. The uniformity of simulation and runtime code is a key design feature of PIER that has enormously improved our ability to debug the system and to experiment with scalability. Moreover, we found that Java did not handle a large number of threads efficiently[2]. Finally, as a matter of taste we found it easier to code using only one thread for event handling.

As a consequence of having only a single main thread, each event handler in the system must complete relatively quickly compared to the inter-arrival rate of new events. In practice this means that handlers cannot make synchronous calls to potentially blocking routines such as network and disk I/O. Instead, the system must utilize asynchronous (a.k.a. "split-phase" or "non-blocking") I/O, registering "callback" routines that handle notifications that the operation is complete[3]. Similarly, any long chunk of CPU-intensive code must yield the processor after some time, by scheduling its own continuation as a timer event. A handler must manage its own state on the heap, because the program stack is cleared after each event yields back to the scheduler. All events originate with the expiration of a timer or with the completion of an I/O operation.

[2]We do not take a stand on whether scalability in the number of threads is a fundamental limit [70] or not [67]. We simply needed to work around Java's current limitations in our own system.

[3]Java does not yet have adequate support for non-blocking file and JDBC I/O operations. For scenarios where these "devices" are used as data sources, we spawn a new thread that blocks on the I/O call and then enqueues the proper event on the Main Scheduler's event priority queue when the call is complete.

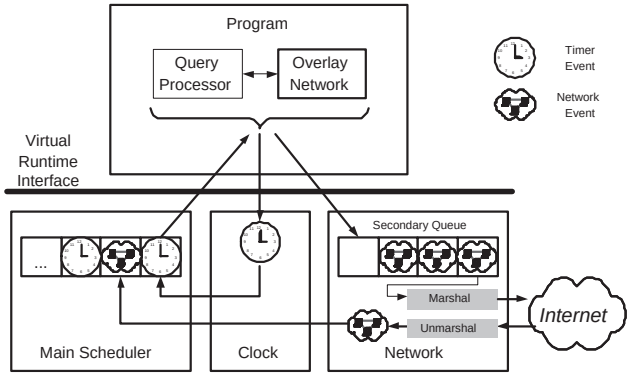

Figure 3: Physical Runtime Environment - A single priority queue in the Main Scheduler stores all events waiting to be handled. Events are enqueued either by setting a timer or through the arrival of a network message. Outbound network messages are enqueued for asynchronous processing. A second I/O thread is responsible for dequeuing and marshaling the messages, and placing them on the network. The I/O thread also receives raw network messages, unmarshals the contents, and places the resulting event in the Main Scheduler's queue.

3.1.3 Physical Runtime Environment

The Physical Runtime Environment consists of the standard system clock, a priority queue of events in the Main Scheduler, an asychronous I/O thread, and a set of IP-based networking libraries (Figure 3). While the clock and scheduler are fairly simple, the networking libraries merit an overview.

UDP is the primary transport protocol used by PIER, mainly due its low cost (in latency and state overhead) relative to TCP sessions. However, UDP does not support delivery acknowledgments or congestion control. To overcome these limitations, we utilize the UdpCC library [60], which provides for acknowledgments and TCP-style congestion control. Although UdpCC tracks each message and provides for reliable delivery (or notifies the sender on failure), it does not guarantee in-order message delivery. TCP sessions are primarily used for communication with user clients. TCP facilitates compatiblity with standard clients and has less problems passing through firewalls and NATs.

3.1.4 Simulation Environment

The Simulation Environment is capable of simulating thousands of virtual nodes on a single physical machine, providing each node with its own independent logical clock and network interface (Figure 4). The Main Scheduler for the simulator is designed to coordinate the discrete-event simulation by demultiplexing events across multiple logical nodes. The program code for each node remains the same in the Simulation Environment as in the Physical Runtime Environment.

The network is simulated at message-level granularity rather than packet-level for efficiency. In other words, each simulated "packet" contains an entire application message

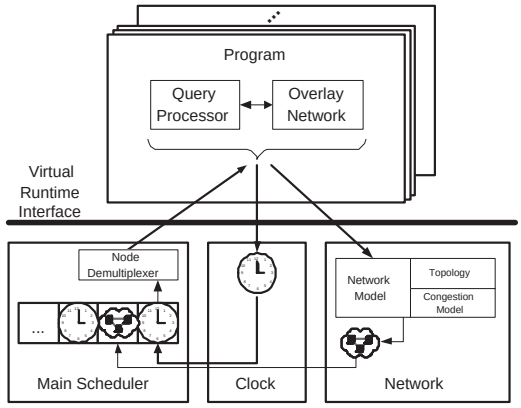

Figure 4: Simulation Environment - The simulator uses one Main Scheduler and priority queue for all nodes. Simulated events are annotated with virtual node identifiers that are used to demultiplex the events to appropriate instances of the Program objects. Outbound network messages are handled by the network model, which uses a topology and congestion model to calculate when the network event should be executed by the program. Some congestion models may reschedule an event in the queue if another outbound message later affects the calculation.

and may be arbitrarily large. By avoiding the need to fragment messages into multiple packets, the simulator has fewer units of data to simulate. Message-level simulation is an accurate approximation of a real network as long as messages are relatively close in size to the maximum packet size on the real network (usually 1500 bytes on the Internet). Most messages in PIER are under 2KB.

Our simulator includes support for two standard network topology types (star and transit-stub) and three congestion models (no congestion, fair queuing, and FIFO queuing). The simulator does not currently simulate network loss (all messages are delivered), but it is capable of simulating complete node failures.

3.2 Overlay Network

Having described the underlying runtime environment and its interfaces, we can now begin to describe PIER's program logic. We begin with the overlay network, which is a key abstraction that is used by PIER in a variety of ways that we will discuss in Section 3.3.6.

Internet-scale systems like PIER require robust communication substrates that keep track of the nodes currently participating in the system, and reliably direct traffic between the participants as nodes come and go. One approach to this problem uses a central server to maintain a directory of all the participants and their direct IP addresses (the original "Napster" model, also used in PeerDB [52]). However, this solution requires an expensive, well-administered, highly available central server, placing control of (and liability for) the system in the hands of the organization that administers that central server.

Instead of a central server, PIER uses a decentralized routing infrastructure, provided by an overlay network. Overlay

Inter-Node Operations

void get(namespace, key, callbackClient)
void put(namespace, key, suffix, object, lifetime)
void $send$(namespace, key, suffix, object, lifetime)
void $renew$(namespace, key, suffix, lifetime)
void $handleGet$(namespace, key, objects[])

Intra-Node Operations

void $localScan$(callbackClient)
void $newData$(callbackClient)
void $upcall$(callbackClient)
void $handleLScan$(namespace, key, object)
void $handleNewData$(namespace, key, object)
continueRouting $handleUpcall$(namespace, key, object)

Table 2: Selected methods provided by the overlay wrapper.

networks are a means of inserting a distributed layer of indirection above the standard IP network. DHTs are a popular class of overlay networks that provide location independence by assigning every node and object an identifier in an abstract identifier space. The DHT maintains a dynamic mapping from the abstract identifier space to actual nodes in the system.

The DHT provides, as its name implies, a hash-table like interface where the hash buckets are distributed throughout the network. In addition to a distributed implementation of a hash table's traditional get and put methods, the DHT also provides additional object access and maintenance methods. We proceed to describe the DHT's three core components – naming, routing, and state – as well as the various DHT interfaces (Table 2).

3.2.1 Naming

Each PIER object in the DHT is named using three parts: a namespace, partitioning key, and suffix. The DHT computes an object's *routing identifier* using the namespace and partitioning key; the suffix is used to differentiate objects that share the same routing identifier. The query processor uses the namespace to represent a table name or the name of a partial result set in a query. The partitioning key is generated from one or more relational attributes used to index the tuple in the DHT (the hashing attributes). Suffixes are tuple "uniquifiers", chosen at random to minimize the chance of a spurious name collision within a table.

3.2.2 Routing

One of the key features of DHTs is their ability to handle churn in the set of member nodes. Instead of a centralized directory of nodes in the system, each node keeps track of a selected set of "neighbors", and this neighbor table must be continually updated to be consistent with the actual membership in the network. To keep this overhead low, most DHTs are designed so that each node maintains only a few neighbors, thus reducing the volume of updates. As a consequence, any given node can only route directly to a handful of other nodes. To reach arbitrary nodes, multi-hop routing is used.

In multi-hop routing, each node in the DHT may be required to forward messages for other nodes. Forwarding entails deciding the next hop for the message based on its destination identifier. Most DHT algorithms require that the message makes "forward progress" at each hop to prevent routing cycles. The definition of "forward progress" is a key differentiator among the various DHT designs; a full discussion is beyond the scope of this paper.

A useful side effect of multi-hop routing is the ability of nodes along the forwarding path to intercept messages before forwarding them to the next hop. Via an upcall from the DHT, the query processor can inspect, modify or even drop a message. Upcalls play an important role in various aspects of efficient query processing, as we will discuss in Section 3.3.

3.2.3 Soft State

Recall that the system does not support persistent storage; instead it places the burden of ensuring persistence on the originator of an object (its *publisher*) using *soft state*, a key design principle in Internet systems [12].

In soft state, a node stores each item for a relatively short time period, the object's "soft-state lifetime", after which the item is discarded. If the publisher wishes to keep an object in the system for longer, it must periodically "renew" the object, to extend its lifetime.

If a DHT node fails, any objects stored at that node will be lost and no longer available to the system. When the publisher attempts to renew the object, its identifier will be handled by a different node than before, which will not recognize the object identifier, cause the renewal to fail, and the publisher must publish the item again, thereby making it available to the system again. Soft-state also has the side-effect of being a natural garbage collector for data. If the publisher fails, any objects published will eventually be discarded.

The choice of a soft-state lifetime is given to the publisher, with the system enforcing a maximum lifetime. Shorter lifetimes require more work by the publisher to maintain persistence, but increase object availability, since failures are detected and fixed by the publisher faster. Longer lifetimes are less work for the publisher but failures can go undetected for longer. The maximum lifetime protects the system from having to expend resources storing an object whose publisher failed long ago.

3.2.4 Implementation

Our overlay network is composed of three modules, the router, object manager, and wrapper (see Figure 5).

The router contains the peer to peer overlay routing protocol, of which there are many options. We currently use Bamboo [60], although PIER is agnostic to the actual algorithm, and has used other DHTs in the past.

As listed in Table 2 the DHT supports a collection of inter-node and intra-node operations. The hash table functionality is provided by a pair of asynchronous inter-node methods, put and get. Both are two-phase operations: first a lookup is performed to determine the identifier-to-IP address mapping, then a direct point-to-point IP communication is used

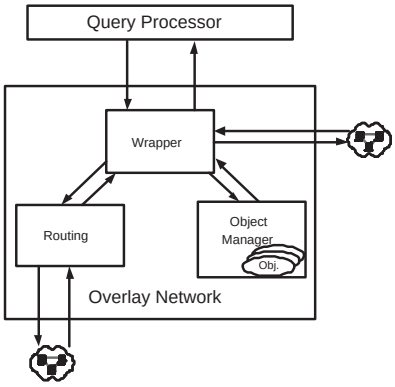

Figure 5: The overlay network is composed of the router, object manager and wrapper. Both the router and wrapper exchange messages with other nodes via the network. The query processor only interacts with the wrapper, which in turn manages the choreography between the router and object manager to fulfill the request.

to perform the operation. When the get operation completes, the DHT passes the data to the query processor through the $handleGet$ callback. The API also supports a lightweight variant of put called $renew$ to renew an object's soft-state lifetime. The $renew$ method can only succeed if the item is already at the destination node; otherwise, the $renew$ will fail and a put must be performed. The $send$ method is similar to a put, except upcalls are provided at each node along the path to the destination. Figure 6 shows how each of the operations is performed.

The two intra-node operations are also key to the query processor. $localScan$ and $handleLScan$ allows the query processor to view all the objects that are present at the local node. $newData$ and $handleNewData$ enable the query processor to be notified when a new object arrives at the node. Finally $upcall$ and $handleUpcall$ allow the query processor to intercept messages sent via the $send$ call.

3.3 Query Processor

Having described the runtime environment and the overlay network, we can now turn our attention to the processing of queries in PIER. We introduce the PIER query processor by first describing its data representation and then explaining the basic sequence of events for executing a query.

3.3.1 Data Representation and Access Methods

Recall that PIER does not maintain system metadata. As a result, every tuple in PIER is self-describing, containing its table name, column names, and column types.

PIER utilizes Java as its type system, and column values are stored as native Java objects. Java supports arbitrarily complex data types, including nesting, inheritance and polymorphism. This provides natural support for extensibility in the form of abstract data types, though PIER does not interpret these types beyond accessing their Java methods.

Tuples enter the system through access methods, which can contact a variety of sources (the internal DHT, remote

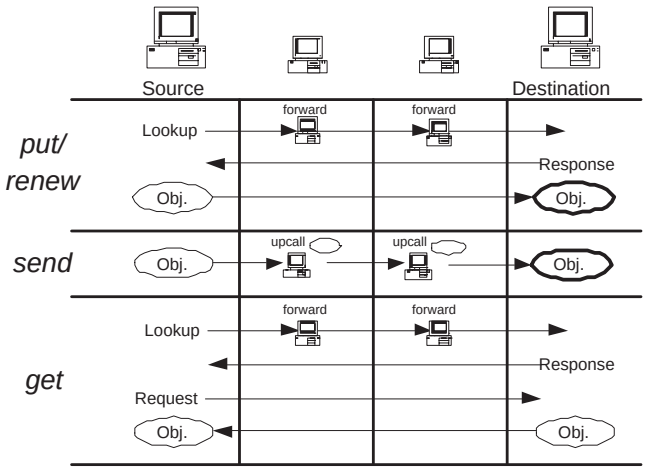

Figure 6: put and $renew$ perform a lookup to find the object's identifier-to-IP mapping, after which they can directly forward the object to the destination. $send$ is very similar to a put, except the object is routed to the destination in a single call. While $send$ uses fewer network messages, each message is larger since it includes the object. get is done via a lookup followed by a request message and finally a response including the object(s) requested.

web pages, files, JDBC, BerkeleyDB, etc.) to fetch the data. The access method converts the data's native format into PIER's tuple format and injects the tuple in the dataflow (Section 3.3.5). Any necessary type inference or conversion is performed by the access method. Unless specified explicitly as part of the query, the access method is unable to perform type checking; instead, type checking is deferred until further in the processing when a comparison operator or function access the value.

3.3.2 Life of a Query

We first overview the life of a query in PIER, and in subsequent sections delve somewhat more deeply into the details.

For PIER we defined a native algebraic ("box and arrow") dataflow language we call UFL[4]. UFL is in the spirit of stream query systems like Aurora [1], and router toolkits like Click [38]. UFL queries are direct specifications of physical query execution plans (including types) in PIER, and we will refer to them as query plans from here on. A graphical user interface called Lighthouse is available for more conveniently "wiring up" UFL[5]. PIER supports UFL graphs with cycles, and such recursive queries in PIER are the topic of research beyond the scope of this paper [42].

An UFL query plan is made up of one or more operator graphs (opgraphs). Each individual opgraph is a connected set of dataflow operators (the nodes) with the edges specifying dataflow between operators (Section 3.3.5). Each operator is associated with a particular implementation.

[4]Currently UFL stands for the "Unnamed Flow Language".

[5]Somewhat to our surprise, many of our early users requested a SQL-like query language. We have implemented a naive version of this functionality, but this interface raises various query optimization issues (Section 4.2).

Separate opgraphs are formed wherever the query redistributes data around the network and the usual local dataflow channels of Section 3.3.5 are not used between sets of operators (similar to where a distributed Exchange operator [24] would be placed). Instead a producer and a consumer in two separate opgraphs are connected using the DHT (actually, a particular namespace within the DHT) as a rendezvous point. Opgraphs are also the unit of dissemintation (Section 3.3.3), allowing different parts of the query to be selectively sent to only the node(s) required by that portion of the query.

After a query is composed, the user application (the client) establishes a TCP connection with any PIER node. The PIER node selected serves as the proxy node for the user. The proxy node is responsible for query parsing and dissemination, and for forwarding results to the client application.

Query parsing converts the UFL representation of the query into Java objects suitable for the query executor. The parser does not need to perform type inference (UFL is a typed syntax) and cannot check the existence or type of column references since there is no system catalog.

Once the query is parsed, each opgraph in the query plan must be disseminated to the nodes needed to process that portion of the query (Section 3.3.3). When a node receives an opgraph it creates an instance of each operator in the graph (Section 3.3.4), and establishes the dataflow links (Section 3.3.5) between the operators.

During execution, any node executing an opgraph may produce an answer tuple. The tuple (or batches of tuples) is forwarded to the client's proxy node. The proxy then delivers the tuple to the client's application.

A node continues to execute an opgraph until a timeout specified in the query expires. Timeouts are used for both snapshot and continuous queries. A natural alternative for snapshot queries would be to wait until the dataflow delivers an EOF (or similar message). This has a number of problems. First, in PIER, the dataflow source may be a massively distributed data source such as the DHT. In this case, the data may be coming from an arbitrary subset of nodes in the entire system, and the node executing the opgraph would need to maintain the list of all live nodes, even under system churn. Second, EOFs are only useful if messages sent over the network are delivered in-order, a guarantee our message layer does not provide. By contrast, timeouts are simple and applicable to both snapshot and continuous queries. The burden of selecting the proper timeout is left to the query writer.

Given this overview, we now expand upon query dissemination and indexing, the operators, dataflow between the operators, and PIER's use of the overlay network.

3.3.3 Query Dissemination and Indexing

A non-trivial aspect of a distributed query system is to efficiently disseminate queries to the participating nodes. The simplest form of query dissemination is to broadcast each opgraph to every node. Broadcast (and the more specialized multicast) in a DHT has been studied by many others [10, 58]. The method we describe here is based upon the distribution tree techniques presented in [10].

PIER maintains a *distribution tree* for use by all queries; multiple trees can be supported for reliability and load balancing. Upon joining the network, each PIER node routes a message (using $send$) containing its node identifier toward a well-known root identifier that is hard-coded in PIER. The node at the first hop receives an upcall with the message, records the node identifier contained in the message, and drops the message. This process creates a tree, where each message informs a parent node about a new child. A node's depth in the tree is equivalent to the number of hops its message would have taken to reach the root. The shape of the tree (fanout, height, imbalance) is dependent on the DHT's routing algorithm[6]. The tree is maintained using soft-state, so periodic messages allow it to adapt to membership changes.

To broadcast an opgraph, the proxy node forwards it to a hard-coded ID for the root of the distribution tree. The root then sends a copy to each "child" identifier it had recorded from the previous phase, which then forwards it on recursively.

Broadcasting is not efficient or scalable, so whenever possible we want to send an opgraph to only those nodes that have tuples needed to process the query. Just like a DBMS uses a disk-based index to read the fewest disk blocks, PIER can use distributed indexes to determine the subset of network nodes needed based on a predicate in an opgraph. In this respect query dissemination is really an example of a distributed indexing problem[7].

PIER currently has three kinds of indexes: a true-predicate index, an equality-predicate index, and a range-predicate index. The true-predicate index is the distribution tree described above: it allows a query that ranges over all the data to find all the data. Equality predicates in PIER are directly supported by the DHT: operations that need to find a specific value of a partitioning key can be routed to the relevant node using the DHT. For range search, PIER uses a new technique called a *Prefix Hash Tree* (PHT), which makes use of the DHT for addressing and storage. The PHT is essentially a resilient distributed trie implemented over DHTs. A full description of the PHT algorithm can be found in [59]. While PHTs have been implemented directly on our DHT codebase, we have yet to integrate them into PIER. The index facility in PIER is extensible, so additional indexes (that may or may not use the DHT) can be also supported in the future.

Note that a primary index in PIER is achieved by publishing a table into the DHT or PHT with the partitioning attributes serving as the index key. Secondary indexes are also possible to create: they are simply tables of (*index-key, tupleID*) pairs, published with *index-key* as the partitioning key. The *tupleID* has to be an identifier that PIER can use to access the tuple (e.g., a DHT name). PIER provides no automated logic to maintain consistency between the secondary index and the base tuples.

In addition to the query dissemination problem described above, PIER also uses its distributed indexing facility in manners more analogous to a traditional DBMS. PIER can use a primary index as the "inner" relation of a Fetch Matches join [44], which is essentially a distributed index join. In this case, each call to the index is like disseminating a small single-table subquery within the join algorithm. Finally, PIER can be used to take advantage of what we have called secondary indexes. This is achieved by a query explicitly specifying a semi-join between the secondary index and the original table; the index serves as the "outer" relation of a Fetch Matches join that follows the *tupleID* to fetch the correct tuples from the correct nodes. Note that this semi-join can be situated as the inner relation of a Fetch Matches join, which achieves the effect of a distributed index join over a secondary index.

3.3.4 Operators and Query Plans

PIER is equipped with 14 logical operators and 26 physical operators (some logical operators have multiple implementations).

Most of the operators are similar to those in a DBMS, such as selection, projection, tee, union, join, group-by, and duplicate elimination. PIER also uses a number of non-traditional operators, including many of our access methods, a result handler, in-memory table materializer, queues, put (similar to Exchange), Eddies [2] (Section 4.2), and a control flow manager (Section 3.3.5).

Join algorithms in PIER include Symmetric Hash join [71] and Fetch Matches join [44]. Common rewrite strategies such as Bloom join and semi-joins can be constructed. In [32] we examine the different join strategies and their trade-offs.

While for the most part PIER's operators and queries are similar to other systems, there are some salient differences:

- **Hierarchical Aggregation.** The simplest method for computing an aggregate is to collect all the source tuples in one location. However if there are many tuples, communication costs could easily overwhelm the one node receiving the data. Instead we want to distribute the *inbandwidth* load to multiple nodes.

  One such method is to have each node compute the partial aggregate for its values and those from a group of other nodes. Instead of explicitly grouping nodes, we can arrange the nodes into a tree following the same process used in query broadcasting (see Section 3.3.3). Each node computes its local aggregate and uses the DHT $send$ call to send it to a root identifier specified in the query. At the first hop along the routing path, PIER receives an upcall, and combines that partial aggregate with its own data. After waiting for more data to arrive from other nodes, the node then forwards the partial aggregate one more hop closer to the root. Eventually the root will receive partial aggregates that include data from every node and the root can produce the answer.

  In the optimal case, each node sends exactly one partial aggregate. To achieve the optimal, each node must know when it has received data from each of it children.

[6] For example, Chord [63] produces distribution trees that are (roughly) binomial; Koorde [35] produces trees that are (roughly) balanced binary.

[7] We do not discuss the role of node-local indexes that enable fast access to data stored at that node. PIER does this in a traditional fashion, currently using main-memory hashtables.

This has problems similar to our discussion of EOF in Section 3.3.2. We discuss this in more detail in [31].

This procedure works well for distributive and algebraic aggregates where only constant state is needed at each step regardless of the amount of source data being aggregated. Holistic aggregates are unlikely to benefit from hierarchical computation.

- **Hierarchical Joins.** Like hierarchical aggregation, the goal of hierarchical joins is to reduce the communication load. In this case, we can reduce the *out-bandwidth* of a node rather than the in-bandwidth.

  In the partitioning ("rehash") phase of a parallel hash join, source tuples can be routed through the network (using $send$), destined for the correct hash bucket on some node. As each tuple is forwarded along the path, each intermediate node intercepts it, caches a copy, and annotates it with its local node identifier before forwarding it along. When two tuples cached at the same node can be joined, and were not previously annotated with a matching node identifier, the join result is produced and sent directly to the proxy. In essence, join results are produced "early". This both improves latency, and shifts the out-bandwidth load from the node responsible for a hash-bucket to nodes along the paths to that node.

  Hierarchical joins only reduce out-bandwidth for some nodes. In particular, when a skewed workload causes one or more hash buckets to receive a majority of the tuples, the network bottleneck at those skewed nodes is ameliorated by offloading out-bandwidth to nodes "along the way". The in-bandwidth at a node responsible for a particular hash bucket will remain the same since it receives every join input tuple that it would have without hierarchical processing.

- **Malformed Tuples.** In a wide-area decentralized application it is likely that PIER will encounter tuples that do not match the schema expected by a query. PIER uses a "best effort" policy when processing such data. Query operators attempt to process each tuple, but if a tuple does not contain the field of the proper type specified in the query, the tuple is simply discarded. Likewise if a comparison (or in general any function) cannot be processed because a field is of an incompatible type, then the tuple is also discarded.

- **No Global Synchronization.** PIER nodes are only loosely synchronized, where the error in synchronization is based on the longest delay between any two nodes in the system at any given time. An opgraph is executed as soon as it is received by a node. Therefore it is possible that one node will begin processing an opgraph and send data to another node that has yet to receive the query. As a consequence, PIER's query operators must be capable of "catching up" when they start, by processing any data that may have already arrived.

- **In-Memory Operators.** Currently, all operators in PIER use in-memory algorithms, with no spilling to disk. This is mainly a result of PIER not having a buffer manager or storage subsystem. While this has not been a problem for many applications, we plan to revisit this decision.

3.3.5 Local Dataflow

Once an opgraph arrives at a node, the local dataflow is set up. A key feature to the design of the intra-site dataflow is the decoupling of the control flow and dataflow within the execution engine.

Recall that PIER's event-driven model prohibits handlers from blocking. As a reuslt, PIER is unable to make use of the widely-used iterator ("pull") model. Instead, PIER adopts a "non-blocking iterator" model that uses pull for control messages, and push for the dataflow. In a query tree, parent operators are connected to their children via a traditional control channel based on function calls. Asynchronous requests for sets of data (*probes*) are issued and flow from parent to child along the graph, much like the $open$ call in iterators. During these requests, each operator sets up its relevant state on the heap. Once the probe has generated state in each of the necessary operators in the opgraph, the stack is unwound as the operators return from the function call initiating the probe.

When an access method receives a probe, it typically registers for a callback (such as from the DHT) on data arrival, or yields and schedules a timer event with the Main Scheduler.

When a tuple arrives at a node via an access method, it is pushed from child to parent in the opgraph via a data channel that is also based on simple function calls: each operator calls its parent with the tuple as an argument. The tuple will continue to flow from child to parent in the plan until either it reaches an operator that removes it from the dataflow (such as a selection), it is consumed by an operator that stores it awaiting more tuples (such as join or group-by), or it enters a queue operator. At that point, the call stack unwinds. The process is repeated for each tuple that matches the probe, so multiple tuples may be pushed for one probe request.

Queues are inserted into opgraphs as places where dataflow processing "comes up for air" and yields control back to the Main Scheduler. When a queue receives a tuple, it registers a timer event (with zero delay). When the scheduler is ready to execute the queue's timer event, the queue continues the tuple's flow from child to parent through the opgraph.

An arbitrary tag is assigned to each probe request. The same tag is then sent with the data requested by that probe. The tag allows for arbitrary reordering of nested probes while still allowing operators to match the data with their stored state (in the iterator model this is not needed since at most one *get-next* request is outstanding on each dataflow edge.

3.3.6 Query Processing Uses of the Overlay

PIER is unique in its aggressive reuse of DHTs for a variety of purposes traditionally served by different components in a DBMS. Here we take a moment to enumerate the various ways the DHT is used.

- **Query Dissemination.** The multi-hop topology in the DHT allows construction of query dissemination trees as described in Section 3.3.3. If a table is published into the DHT with a particular namespace and partitioning key, then the query dissemination layer can route queries with equality predicates on the partitioning key to just the right nodes.

- **Hash Index:** If a table is published into the DHT, the table is essentially stored in a distributed hash index keyed on the partitioning key. Similiarly, the DHT can also be used to create secondary hash indexes.

- **Range Index Substrate:** The PHT technique provides resilient distributed range-search functionality by mapping the nodes of a trie search structure onto a DHT [59].

- **Partitioned Parallelism:** Similar to the Exchange operator [24], the DHT is used to partition tuples by value, parallelizing work across the entire system while providing a network queue and separation of control-flow between contiguous groups of operators (opgraphs).

- **Operator State:** Because the DHT has a local storage layer and supports hash lookups, it is used directly as the main-memory state for operators like hash joins and hash-based grouping, which do not maintain their own separate hashtables.

- **Hierarchical Operators:** The inverse of a dissemination trees is an aggregation tree, which exploits multi-hop routing and callbacks in the DHT to enable hierarchical implementations of dataflow operators (aggregations, joins).

4 Future Work

In this section we discuss the two primary areas of continuing research on PIER: security and query optimization.

4.1 Security

In the last two years we have been running PIER on the PlanetLab testbed, now over 350 machines on 5 continents. That environment raises many real-world challenges of scale and reliability, but remains relatively benign in terms of the participants in the system. Our next goal is to ready PIER for truly Internet-scale deployment "in the wild," which will require facing the interrelated questions of security and robustness. In this section we present our initial considerations in this direction, identifying major challenges and techniques we are considering to meet these challenges.

4.1.1 Challenges

Many of challenges facing loosely-coupled distributed systems that operate in unfriendly environments have been identified before. In the context of peer-to-peer systems, Wallach gives a good survey [68]. We concentrate here on challenges particular to our query processing system.

Result fidelity is a measure of how close a returned result is to the "correct" result. Depending on the query, fidelity can be related to information retrieval success metrics such as precision and recall, or to the numerical accuracy of a computed result such as an aggregation function. Fidelity may deteriorate due to both network failures (message loss between nodes, or nodes becoming unreachable) and node failures (causing partial or total loss of data and results on a node). Both classes of failures can be caused by malicious activity, such as network denial-of-service (DoS) attacks or node break-ins. In addition, a compromised or malicious node can reduce fidelity through *data poisoning*, for instance by introducing an outlier value that derails the computation of a minimum.

Resource management in a highly-distributed, loosely coupled system like PIER faces additional challenges beyond the traditional issues of fairness, timeliness, etc. Important concerns include *isolation*, *free-riding*, *service flooding*, and *containment*. Isolation prevents client-generated tasks from causing the distributed system to consume vastly more resources than required at the expense of other tasks, for example due to poorly or maliciously constructed query plans. Free-riding nodes in peer-to-peer systems exploit the system while contributing little or no resources. Service flooding by malicious clients is the practice of sending many requests to a service with the goal of overloading it. Finally, containment is important to avoid the use of the powerful PIER infrastructure itself as a launchpad for attacks against other entities on the Internet; consider, for instance, a slew of malicious queries all of which name an unsuspecting victim as the intended recipient for their (voluminous) results.

Accountability of components in PIER, that is, the ability to assign responsibility for incorrect or disruptive behavior, is important to ensure reliable operation. When misbehavior is detected, accountability helps identify the offending nodes and justifies corrective measures. For example, the query can be repeated excluding those nodes (in the short term), or the information can be used as input to a reputation database used for node selection in the future.

Finally, *politics* and *privacy* are ever-present issues in any peer-to-peer system. Different data sources may have differing policy restrictions on their use, including access control (e.g., export controls on data) and perhaps some notion of pricing (e.g., though coarse-grained summaries of a data set may be free, access to the raw data may require a fee). The flip-side of such data usage policies is user privacy. Adoption of large p2p query-based applications may be hindered by end-user concerns about how their query patterns are exploited: such patterns may reveal personal information or help direct targeted fidelity attacks. Certain data sets may also require some anonymity, such as disassociation between data points and the users responsible for those data points.

4.1.2 Defenses

Currently PIER concentrates primarily on mechanics and feasibility and less on the important issues of security and fault tolerance. In this section, we present defensive avenues we are investigating for PIER and, in each case, we outline

the support that PIER already incorporates, the approaches we are actively pursuing but have not yet implemented, and further explorations we foresee for our future work. Wallach [68] and Maniatis et al. [47] survey available defenses within peer-to-peer environments.

- **Redundancy.** Redundancy is a simple but powerful general technique for improving both security and robustness. Using multiple, randomly selected entities to compute the result for the same operator may help to reveal maliciously suppressed inputs as well as overcome the temporary loss of network links. Similarly, multiple overlay paths can be used to thwart malicious attempts to drop, delay, or modify PIER messages in query dissemination or result aggregation.

  The current codebase does not incorporate any of these techniques. We are, however, studying the benefits offered by different dissemination and aggregation topologies in minimizing the influence of an adversary on the computed result. Specifically, we examine the change in simple metrics such as the fraction of data sources suppressed by the adversary and relative result error, and plan in the future to consider more complex ones such as *maximum influence* [37]. We hope that the outcome of this study will help us to adapt mechanism designs for duplicate-insensitive summarization recently proposed for benign environments [3, 13, 50] to our significantly more unfriendly target environment.

- **Rate Limitation.** A powerful defense against the abuse of PIER's resources calls for the enforcement of rate limits on the dissemination and execution of queries, and the transfer of results (e.g., [16, 47]). These rate limits may be imposed on queries by particular clients, e.g., to prevent those clients from unfairly overwhelming the system with expensive operations; they may be imposed on the results traffic directed towards particular destinations, to limit the damage that malicious clients or bugs can cause PIER itself to inflict upon external entities; they may be imposed by one PIER node on another PIER node, to limit the amount of free-riding possible when some PIER nodes are compromised.

  PIER takes advantage of the virtualization inherent in its Java environment to sandbox operators as they execute. This is an important first step before rate limits — or any other limits including access controls, etc. — can be imposed on running operators. Although the system does not currently impose any such limits, we are actively investigating rate limitations against particular clients. We monitor, at each PIER node, the total resource consumption (e.g., CPU cycles, disk space, memory, etc.) of that client's query operators within a time window. When a node detects that this total exceeds a certain threshold, it contacts other PIER nodes to compute the aggregate consumption by the suspect client over the whole system. Given that aggregate, a PIER node can throttle back, primarily via the sandboxes, the amount of local resources it makes available to the culprit's operators. Note, however, that per-client controls such as those we propose are dependent on a dependable authentication mechanism not only for PIER nodes but also for the clients who use them; otherwise, Sybil attacks [19] in which a malicious client uses throwaway identities, one per few queries, never quite reaching the rate limitation thresholds, can nullify some of the benefits of the defense. Assigning client identifiers in a centralized fashion [9] may help with this problem, but reverts to a tightly-coupled subsystem for identity.

  For the scenarios in which PIER nodes themselves may misbehave, e.g., to free-ride, we are hoping to incorporate into the system a coarser-grained dynamic rate-limitation mechanism that we have previously used in a different peer-to-peer problem [47]. In this scheme, PIER nodes can apply a reciprocative strategy [21], by which node A executes a query injected via node B only if B has recently executed a query injected via A. The objective of this strategy is to ensure that A and B maintain a balance of executed queries (and the associated resources consumed) injected via each; A rate-limits queries injected via B according to its own need to inject new queries. Reciprocation works less effectively when the contribution of individual peers in the system varies, for instance due to the popularity of a peer's resources. However, such reciprocative approaches are especially helpful in infrastructural p2p environments such as PIER, in which nodes are expected to interact with each other frequently and over a long time period.

- **Spot-checking and Early Commitment.** Program verification techniques such as *probabilistic spot-checking* [20] and early commitment via authenticated data structures [8, 23, 46, 48] can be invaluable in ensuring the accountable operation of individual components within a loosely-coupled distributed system. In the SIA project [55], such techniques are used to aggregate information securely in the more contained setting of sensor networks. Briefly, when a client requesting a query wishes to verify the correct behavior of the (single) aggregator, it samples its inputs and ensures that these samples are consistent with the result supplied by the aggregator. The client knows that the input samples it obtains are the same as those used by the aggregator to compute its result, because the aggregator commits early on those inputs cryptographically, making it practically impossible to "cover its tracks" after the fact, during spot checking.

  In the context of PIER, the execution of operators, including aggregation operators, is distributed among many nodes, some of which may be malicious. In the near term, we are investigating the use of spot checks, first to verify the correct execution of individual nodes within an aggregation tree, for example, that a sum operator added the inputs of its aggregation children correctly. Second, to ensure that all data inputs are included

in the computation, we trace the computation path from sampled data sources to the query result. Third, to ensure that all included data inputs should, in fact, be included, we sample execution paths through an aggregation tree and verify that the raw inputs come from legitimate sources. A client who detects that a result is suspect — i.e., is inconsistent with subsequent spot checks — can refine the approximation with additional sampling, use redundancy to obtain "a second opinion," or simply abort the query.

In the long term, we hope to implement in PIER the promising principle of "trust but verify" [75], based on mechanisms like this that can secure undeniable evidence certifying the misbehavior of a system participant. Such evidence avoids malicious "framing" by competitors, and can be a powerful tool to address issues of both accountability and data fidelity.

4.2 Query Optimization

PIER's UFL query language places the responsibility for query optimization in the hands of the query author. Our initial assumption was that (a) application designers using PIER would be Internet systems experts, more comfortable with dataflow diagrams like the Click router [38] than with SQL, and (b) traditional complex query optimizations like join ordering were unlikely to be important for massively distributed applications. With respect to the first assumption, we seem simply to have been wrong – many users (e.g., administrators of the PlanetLab testbed) far prefer the compact syntax of SQL to UFL or even the Lighthouse GUI. The second issue is still open for debate. We do see multiway join queries in the filesharing application (each keyword in a query becomes a table instance to be joined), but very few so far in network monitoring. Choices of join algorithms and access methods remain an important issue in both cases.

We currently have an implementation of a SQL-like language over PIER using a very naive optimizer. We have considered two general approaches for a more sophisticated optimizer, which we discuss briefly.

4.2.1 Static Optimization

It is natural to consider the implementation of a traditional R*-style [40] static query optimizer for PIER. An unusual roadblock in this approach is PIER's lack of a metadata catalog. PIER has no mechanism to store statistics on table cardinalities, data distributions, or even indexes. More fundamentally, there is no process to create nor place to store an agreed-upon list of the tables that should be modeled by the metadata!

The natural workaround for this problem is for an end-user p2p application based on PIER to "bake" the metadata storage and interpretation into its application logic. Another related approach is to store metadata outside the boundaries of PIER. Table and column statistics could be computed by running (approximate) aggregation queries in PIER, and the whole batch of statistics could be stored and disseminated in a canonical file format outside the system. This out-of-band approach is taken by the popular BitTorrent p2p filesharing network: users exchange lists of file names ("torrents") on websites, in chatrooms, etc. In the case of PIER, the metadata would of course be far richer. But an analogous approach is possible, including the possibility that different communities of users might share independent catalogs that capture subsets of the tables available.

This design entirely separates the optimizer from the core of PIER, and could be built as an add-on to the system. We are not currently pursuing a static optimization approach to PIER for two reasons. First, we suspect that the properties of a large p2p network will be quite volatile, and the need for runtime reoptimization will be endemic. Second, one of our key target applications – distributed network monitoring – is naturally a multi-user, multi-query environment, in which multi-query optimization is a critical feature. Building a single-query R*-style optimizer will involve significant effort, while only providing a first step in that direction.

4.2.2 Distributed Eddies

In order to move toward both runtime reoptimization and multi-query optimization, we have implemented a prototype version of an eddy [2] as an optional operator that can be employed in UFL plans [33]. A set of UFL operators can be "wired up" to an eddy, and in principle benefit from the eddy's ability to reorder the operators. A more involved implementation (e.g., using SteMs [56] or STAIRS [17]) could also do adaptive selection of access methods and join algorithms, and the TelegraphCQ mechanism for multiquery optimization could be fairly naturally added to the mix [45].

The above discussion is all in the realm of mechanism: these components could all be implemented within the framework of PIER's dataflow model. But the implementation of an intelligent eddy routing policy for a distributed system is the key to good performance, and this has proven to be a challenging issue [33]. Eddies employ two basic functions in a routing policy: the *observation* of dataflow rates into and out of operators, and based on those observations a *decision* mechanism for choosing how to route tuples through operators. In a centralized environment, observation happens naturally because the eddy intercepts all inputs and outputs to each operator. In PIER, each node's eddy only sees the data that gets routed to (or perhaps through) that node. This makes it difficult for a local eddy instance to make good global decisions about routing. Eddies could communicate across sites to aggregate their observations, but if done naively this could have significant overheads. The degree of coordination merited in this regard is an open issue.

5 Related Work

PIER is currently the only major effort toward an Internet-scale relational query system. However it is inspired by and related to a large number of other projects in both the DB and Internet communities. We present a brief overview here; further connections are made in [32].

5.1 Internet Systems

There are two very widely-used Internet directory systems that have simple query facilities. DNS is perhaps the most ubiquitous distributed query system on the Internet. It supports exact-match lookup queries via a hierarchical design in both its data model (Internet node names) and in its implementation, relying on a set of (currently 13) root servers at well-known IP addresses. LDAP is a hierarchical directory system largely used for managing lookup (selection) queries. There has been some work in the database research community on mapping database research ideas into the LDAP domain and vice versa (e.g., [36]). These systems have proved effective for their narrow workloads, though there are persistent concerns about DNS on a number of fronts [51].

As is well known, p2p filesharing is a huge phenomenon, and systems like KaZaA and Gnutella each have hundreds of thousands of users. These systems typically provide simple Boolean keyword query facilities (without ranking) over short file names, and then coordinate point-to-point downloads. In addition to having limited query facilities, they are ineffective in some basic respects at answering the queries they allow; the interested reader is referred to the many papers on the subject (e.g., [41, 73]).

5.2 Database Systems

Of course we owe a debt to early generations of distributed databases as mentioned in Section 1, but in many ways, PIER's architecture and algorithms are closer to parallel Database Systems like Gamma [18], Volcano [24], etc. – particularly in the use of hash-partitioning during query processing. Naturally the parallel systems do not typically worry about distributed issues like multi-hop Internet routing in the face of node churn.

In terms of its data semantics, PIER most closely resembles centralized data integration and web-query systems like Tukwila [34] and Telegraph [61]. Those systems also reached out to data from multiple autonomous sites, without concern for the storage semantics across the sites.

Another point of reference in the database community is the nascent area of distributed stream query processing, an application that PIER supports. The Aurora* proposal [11] focuses on a small scale of distribution within a single administrative domain, with stronger guarantees and support for quality-of-service in query specification and execution. The Medusa project augments this vision with Mariposa-like economic negotiation among a few large agents.

Tian and DeWitt presented analytical models and simulations for distributed eddies [65]. Their work illustrated that the metrics used for eddy routing policies in centralized systems do not apply well in the distributed setting. Their approaches are based on each node periodically broadcasting its local eddy statistics to the entire network, which would not scale well in a system like PIER.

In terms of declarative query semantics for widely distributed systems, promising recent work by Bawa et al. [3] addresses in-network semantics for both one-shot and continuous aggregation queries, focusing on faults and churn during execution. In the PIER context, open issues remain in capturing clock jitter and soft state semantics, as well as complex, multi-operator queries.

5.3 Hybrids of P2P and DB

Gribble, et al. were the first to make the case for a joint research agenda in p2p technologies and database systems [25]. Another early vision of p2p databases was presented by Bernstein et al. [4], who used a medical IT example as motivation for work on what is sometimes called "multiparty semantic mediation": the semantic challenge of integrating many peer databases with heterogeneous schemas. This area is a main focus of the Piazza project; a representative result is their recent work on mediating schemas transitively as queries propagate across multiple databases [27]. From the perspective of PIER and related Internet systems, there are already clear challenges and benefits in unifying the abundant *homogeneous* data on the Internet [32]. These research agendas are complementary with PIER's; it will be interesting to see how the query execution and semantic mediation work intersects over time and the application interfaces defined to connect them. An early effort in this regard is the PeerDB project [52], though it relies on a central directory server, and its approach to schema integration is quite simple.

PIER is not the only system to address distributed querying of data on the Internet. The IrisNet system [22] has similar goals to PIER, but its design is a stark contrast: IrisNet uses a hierarchical data model (XML) and a hierarchical network overlay (DNS) to route queries and data. As a result, IrisNet shares the characteristics of traditional hierarchical databases: it is best used in scenarios where the hierarchy changes infrequently, and the queries match the hierarchy. Astrolabe [66] is another system that focuses on a hierarchy: in this case the hierarchy of networks and sub-networks on the Internet. Astrolabe supports a data-cube-like roll-up facility along the hierarchy, and can only be used to maintain and query those roll-ups.

Another system that shares these goals is Sophia [69], a distributed Prolog system for network information. Sophia's vision is essentially a superset of PIER's, inasmuch as the relational calculus is a subset of Prolog. To date Sophia provides no distributed optimization or execution strategies, the idea being that such functionality is coded in Prolog as part of the query.

A variety of the component operations in PIER are being explored in the Internet systems community. Distributed aggregation has been the focus of much work; a recent paper by Yalagandula and Dahlin [72] is a good starting point, as it also surveys the earlier work. Range indexing is another topic that is being explored in multiple projects (e.g., [14, 6, 26], etc.) We favor the PHT scheme in PIER because it is simpler than most of these other proposals: it reuses the DHT rather than requiring a separate distributed mechanism as in [14], it works over any DHT (unlike Mercury [6]), and it appears to be a good starting point for resiliency, concurrency, and correctness – issues that have been secondary in most of the related work.

6 Conclusions

When we began the design of PIER, we expected that the mixture of networking and database issues and expertise would lead to unusual architectural choices. This has indeed been the case. We anticipated some of these cross-domain issues from the start. For example, PIER's aggressive, multipurpose exercising of the overlay network was of interest to the networking researchers in the group; the design of a router-like (event-driven, push-based) dataflow core for query execution was of interest to the database researchers. Other challenges came as a surprise, including the many ramifications of forgoing metadata storage in a general-purpose query engine, and the interrelationships between query dissemination and index-based access methods.

We believe that the current state of PIER will be a strong basis for our future work, though we still have many remaining questions on the algorithmic and architectural front. In addition to the issues covered in Sections 4.1 and 4.2, these include efficient processing of recursive queries for network routing, high-performance integration of disk-based persistent storage, and the challenges of declarative query semantics in a soft-state system made up of unsynchronized components.

7 Acknowledgments

We would like to thank Shawn Jeffery, Nick Lanham, Sam Mardanbeigi, and Sean Rhea for their help in implementing PIER.

This research was funded by NSF grants IIS-0209108, IIS-0205647, and 5710001486.

References

[1] Daniel J. Abadi, Don Carney, Ugur etintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 261–272, Dallas, May 2000.

[3] Mayank Bawa, Aristides Gionis, Hector Garcia-Molina, and Rajeev Motwani. The Price of Validity in Dynamic Networks. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Paris, June 2004.

[4] Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data Management for Peer-to-Peer Computing : A Vision. In *Proc. of the 5th International Workshop on the Web and Databases (WebDB)*, June 2002.

[5] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. Query Processing in a System for Distributed Databases (SDD-1). In *Proc. of the ACM Transactions of Database Systems*, volume 6, pages 602–625, 1981.

[6] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan Seshan. Mercury: A Scalable Publish-Subscribe System for Internet Games. In *Proc. of the 1st Workshop on Network and System Support for Games*, pages 3–9. ACM Press, 2002.

[7] Eric A. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[8] Ahto Buldas, Peeter Laud, and Helger Lipmaa. Eliminating Counterevidence with Applications to Accountable Certificate Management. *Jounal of Computer Security*, 10(3):273–296, 2002.

[9] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *Proc. of the 5th Usenix Symposium on Operating Systems Design and Implementation*, pages 299–314, Boston, MA, USA, December 2002.

[10] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. In *Proc. of the IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications)*, October 2002.

[11] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[12] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. of the ACM SIGCOMM Conference*, August 1988.

[13] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. Approximate Aggregation Techniques for Sensor Databases. In *Proc. of the International Conference on Data Engineering (ICDE)*, 2004.

[14] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. In *Proc. of the 7th International Workshop on the Web and Databases (WebDB)*, Paris, France, June 2004.

[15] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Canada, October 2001.

[16] Neil Daswani and Hector Garcia-Molina. Query-Flood DoS Attacks in Gnutella. In *Proc. of the ACM Conference on Computer and Communications Security*, pages 181–192, Washington, DC, USA, November 2002.

[17] Amol Deshpande and Joseph M. Hellerstein. Lifting the Burden of History from Adaptive Query Processing. In *Proc. of the 30th International Conference on Very Large Data Bases*, September 2004.

[18] David J. Dewitt, Shahram Ghandeharizadeh, Donovan A. Schneider, Allan Bricker, Hui-I Hsiao, and Rick Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[19] John Douceur. The Sybil Attack. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 251–260, Boston, MA, USA, March 2002.

[20] Funda Ergün, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. In *Proc. of the 13th Annual ACM Symposium on Theory of Computing*, pages 259–268, Dallas, TX, USA, 1998. ACM Press.

[21] Michal Feldman, Kevin Lai, Ion Stoica, and John Chuang. Robust Incentive Techniques For Peer-to-Peer Networks. In *Proc. of the 5th ACM conference on Electronic commerce*, pages 102–111, New York, NY, USA, 2004. ACM Press.

[22] Phillip B. Gibbons, Brad Karp, Yan Ke, Suman Nath, and Srinivasan Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), October-December 2003.

[23] Michael T. Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an Authenticated Dictionary with Skip Lists and Commutative Hashing. In *Proc. of the DARPA Information Survivability Conference and Exposition (DISCEX)*, Anaheim, CA, USA, June 2001.

[24] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, May 1990. ACM Press.

[25] Steven D. Gribble, Alon Y. Halevy, Zachary G. Ives, Maya Rodrig, and Dan Suciu. What Can Databases Do for Peer-to-Peer? In *Proc. of the 4th International Workshop on the Web and Databases (WebDB)*, Santa Barbara, May 2001.

[26] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate Range Selection Queries in Peer-to-peer. In *Proc. of the First Biennial Conference on Innovative Data Systems Research*, January 2003.

[27] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema Mediation in Peer Data Management Systems. In *Proc. of the 19th International Conference on Data Engineering (ICDE)*, Bangalore, India, 2003.

[28] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Thau Loo, Scott Shenker, and Ion Stoica. Complex Queries in DHT-based Peer-to-Peer Networks. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[29] Joseph M. Hellerstein. Toward Network Data Independence. *SIGMOD Record*, 32, September 2003.

[30] Jeff Hodges and RL Bob Morgan. Lightweight Directory Access Protocol (v3): Technical Specification, September 2002.

[31] Ryan Huebsch, Brent Chun, and Joseph M. Hellerstein. PIER on PlanetLab: Initial Experience and Open Problems. Technical Report IRB-TR-03-043, Intel Research Berkeley, November 2003.

[32] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker, and Ion Stoica. Querying the Internet with PIER . In *Proc. of the 29th International Conference on Very Large Data Bases*, September 2003.

[33] Ryan Huebsch and Shawn R. Jeffery. FREddies: DHT-Based Adaptive Query Processing via FedeRated Eddies. Technical Report UCB/CSD-04-1339, UC Berkeley, July 2004.

[34] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data Integration. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Philadelphia, PA, June 1999.

[35] M. Frans Kaashoek and David Karger. Koorde: A simple degree-optimal distributed hash table. In *Proc. of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, Berkeley, CA, February 2003.

[36] Olga Kapitskaia, Raymond T. Ng, and Divesh Srivastava. Evolution and Revolutions in LDAP Directory Caches. In *Proc. of the International Conference on Extending Database Technology (EDBT)*, pages 202–216, Konstanz, Germany, March 2000.

[37] David Kempe, Jon Kleinberg, and Éva Tardos. Maximizing the Spread of Influence Through a Social Network. In *Proc. of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146. ACM Press, August 2003.

[38] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[39] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, November 2000.

[40] Guy Lohman, C. Mohan, Laura M. Haas, B. G. Lindsay, Paul F. Wilms, and Dean Daniels. Query Processing in R*. Technical Report R.14272, IBM Research Report, April 1984.

[41] Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Scott Shenker, and Ion Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. In *Proc. of the 30th International Conference on Very Large Data Bases*, September 2004.

[42] Boon Thau Loo, Joseph M. Hellerstein, and Ion Stoica. Customizable Routing with Declarative Queries. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[43] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. The Case for a Hybrid P2P Search Infrastructure. In *Proc. of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, San Diego, CA, February 2004.

[44] Lothar F. Mackert and Guy M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *Proc. of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 149–159, Kyoto, August 1986.

[45] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously Adaptive Continuous Queries. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Madison, June 2002.

[46] Petros Maniatis and Mary Baker. Secure History Preservation Through Timeline Entanglement. In *Proc. of the 11th USENIX Security Symposium*, pages 297–312, San Francisco, CA, USA, August 2002.

[47] Petros Maniatis, TJ Giuli, Mema Roussopoulos, David S. H. Rosenthal, and Mary Baker. Impeding Attrition Attacks in P2P Systems. In *Proc. of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004. ACM SIGOPS.

[48] Ralph C. Merkle. Protocols for Public Key Cryptosystems. In *Proc. of the Symposium on Security and Privacy*, pages 122–133, Oakland, CA, U.S.A., April 1980. IEEE Computer Society.

[49] Paul Mockapetris. Domain names – implementation and specification, November 1987.

[50] Suman Nath, Phillip Gibbons, Zachary Anderson, and Srinivasan Seshan. Synopsis Diffusion for Robust Aggregation in Sensor Networks. In *Proc. of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, USA, November 2004.

[51] Internet Navigation and the Domain Name Systems: Technical Alternatives and Policy Implications, 2004. `http://www.nationalacademies.org/dns`.

[52] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *Proc. of the 19th International Conference on Data Engineering (ICDE)*, Bangalore, India, March 2003.

[53] The Network Simulator - ns2, 2004. `http://www.isi.edu/nsnam/ns/index.html`.

[54] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. of the 1st ACM Workshop on Hot Topics in Networks (HotNets)*, Princeton, October 2002.

[55] Bartosz Przydatek, Dawn Song, and Adrian Perrig. SIA: Secure Information Aggregation in Sensor Networks. In *Proc. of the 2nd ACM Conference on Embedded Network Sensor Systems (SenSys)*, Los Angeles, CA, USA, November 2004.

[56] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proc. of the 19th International Conference on Data Engineering (ICDE)*, Bangalore, India, March 2003.

[57] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content Addressable Network. In *Proc. of the ACM SIGCOM Conference*, Berkeley, CA, August 2001.

[58] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker. Application-level Multicast using Content-Addressable Networks. In *Proc. of the 2nd International Workshop of Network Group Communication (NGC)*, 2001.

[59] Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Research Berkeley, June 2003.

[60] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *Proc. of the USENIX Annual Technical Conference (USENIX)*, Boston, Massachusetts, June 2004.

[61] Mehul A. Shah, Samuel R. Madden, Michael J. Franklin, and Joseph M. Hellerstein. Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System. *ACM SIGMOD Record*, 30, December 2001.

[62] Ambuj Shatdal and Jeffrey F. Naughton. Adaptive Parallel Aggregation Algorithms. In *Proc. of the ACM SIGMOD international conference on Management of data*, pages 104–114. ACM Press, 1995.

[63] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proc. of the ACM SIGCOMM Conference*, pages 149–160, 2001.

[64] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 5(1):48–63, 1996.

[65] Feng Tian and David J. DeWitt. Tuple Routing Strategies for Distributed Eddies. In *Proc. of the 29th International Conference on Very Large Data Bases*, September 2003.

[66] Robbert van Renesse, Kenneth P. Birman, Dan Dumitriu, and Werner Vogel. Scalable Management and Data Mining Using Astrolabe. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[67] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 268–281. ACM Press, 2003.

[68] Dan Wallach. A Survey of Peer-to-Peer Security Issues. In *Proc. of the International Symposium on Software Security*, 2002.

[69] Mike Wawrzoniak, Larry Peterson, and Timothy Roscoe. Sophia: An Information Plane for Networked Systems. In *Proc. of the 2nd ACM Workshop on Hot Topics in Networks (HotNets)*, MA, USA, November 2003.

[70] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of the 18th ACM symposium on Operating systems principles (SOSP)*, pages 230–243. ACM Press, 2001.

[71] Annita N. Wilschut and Peter M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. of the 1st International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 68–77, 1991.

[72] Praveen Yalagandula and Mike Dahlin. SDIMS: A Scalable Distributed Information Management System. In *Proc. of the ACM SIGCOMM Conference*, Portland, Oregon, 2004.

[73] Beverly Yang and Hector Garcia-Molina. Improving Search in Peer-to-Peer Systems. In *Proc. of the 22nd International Conference on Distributed Computing Systems (ICDCS)*, Vienna, Austria, July 2002.

[74] Vinod Yegneswaran, Paul Barford, and Somesh Jha. Global Intrusion Detection in the DOMINO Overlay System. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, February 2004.

[75] Aydan R. Yumerefendi and Jeffrey Chase. Trust but Verify: Accountability for Internet Services. In *Proc. of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004. ACM SIGOPS.