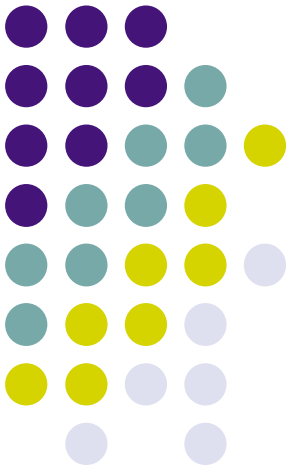
Querying the Internet with PIER

("Peer-to-Peer Information Exchange and Retrieval")

Ryan Huebsch
Nick Lanham
Scott Shenker

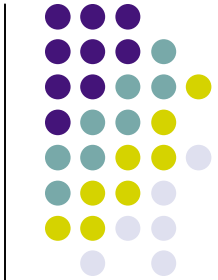
Joseph Hellerstein
Boon Thau Loo
Ion Stoica



CS856

Nabeel Ahmed

Outline



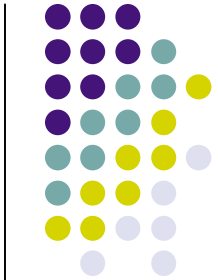
- Motivation
- Architecture
- Evaluation
- Current Status
- Future work
- References
- Discussion

Why do we need PIER?

- P2P systems among fastest-growing technologies in computing
  - Lots of *potential* applications
  - *Hot area* with growing research community
- Deployed P2P network limitations
  - Poor Scaling
  - Impoverished Query Languages
- Traditional dist. system principles not suitable
  - Main problem: *degree of distribution*

DHTS!

Goals for PIER



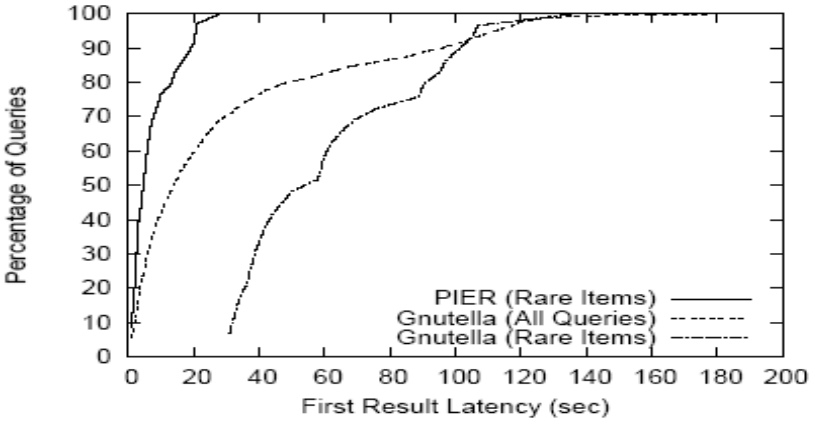
- PIER Agenda:
  - "Technology Push": Massively distributed query processing
  - "Application Pull": Query internet data without database design and maintenance/integration
- Contributions:
  - Architecture *marrying* P2P systems with traditional query processing systems
  - Flexible framework for wide variety of applications
  - Evaluation confirming scalability of system

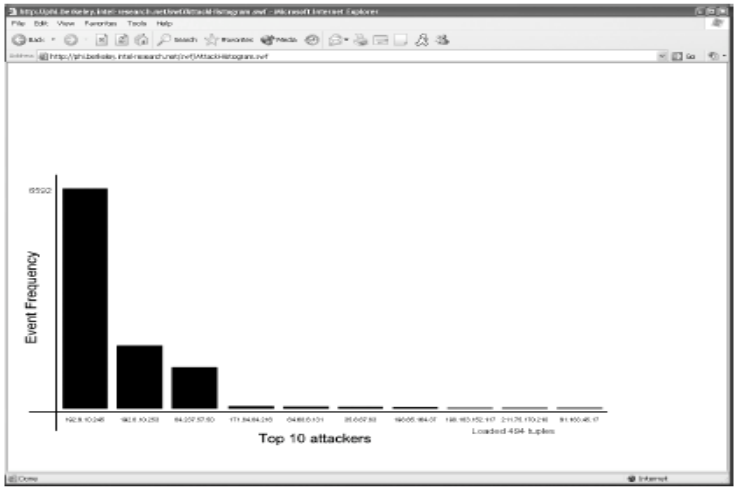
Key Contribution: "First serious treatment of scalability for relational querying engines at such massive scales"

Applications

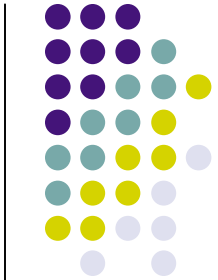
- P2P File Sharing
  - Integrated with Gnutella (Hybrid Infrastructure)
  - Rare Items found using PIER engine
  - Improved recall by 18%!
- Endpoint Network Monitoring
  - Idea: Provide network Intrusion detection
  - Using standard *network* schemas
  - Flexibly and scalably share information

    ("Communal Security")



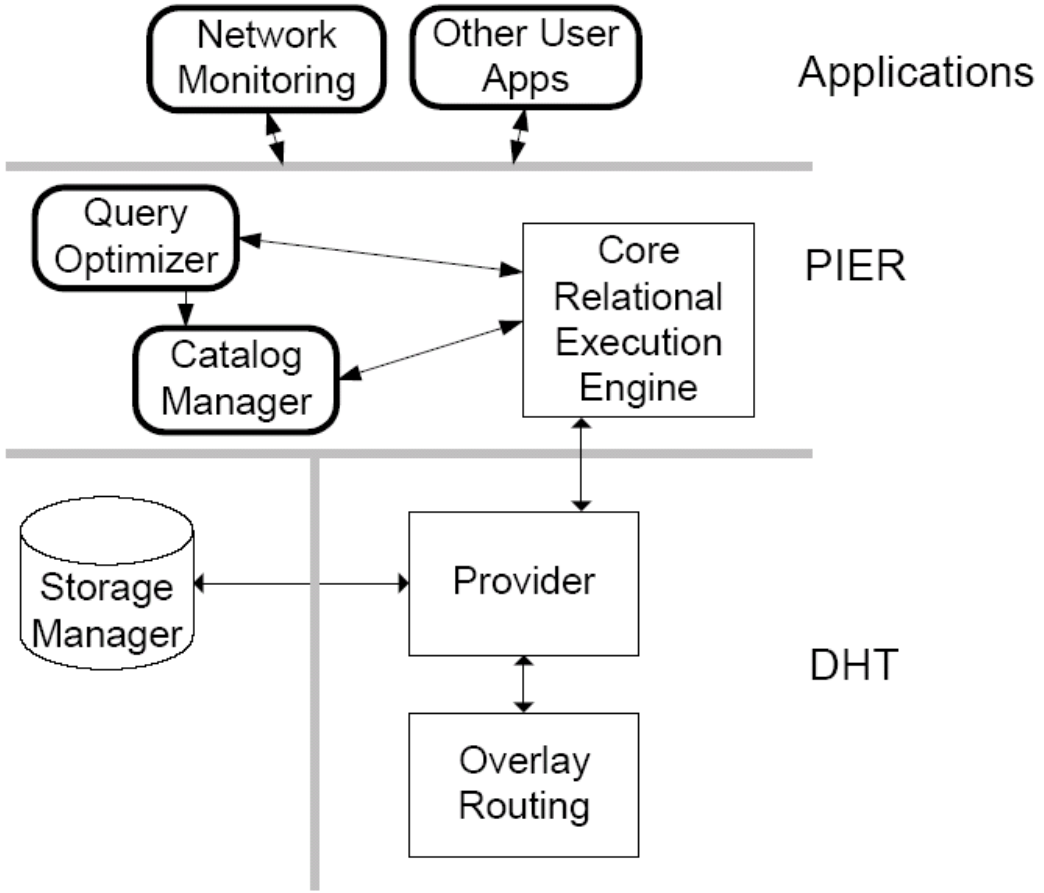


“Relaxed” Design Principles

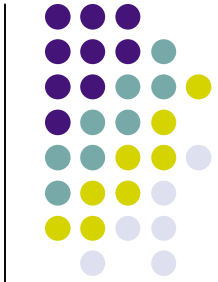


- Relaxed Consistency
  - Traditional ACID transactions severely limit scalability/availability of distributed databases (*CAP conjecture*)
  - Provide best-effort results
- Organic Scaling
  - Avoid architectures that require *a priori* knowledge of network
- Natural habitat
  - No CREATE TABLE/INSERT
  - No “publish to web server”
  - Wrappers or gateways allow the information to be accessed where it is created
- Standard Schemas via Grassroots software
  - Data is produced by widespread software providing a de-facto schema to utilize

Architecture of PIER

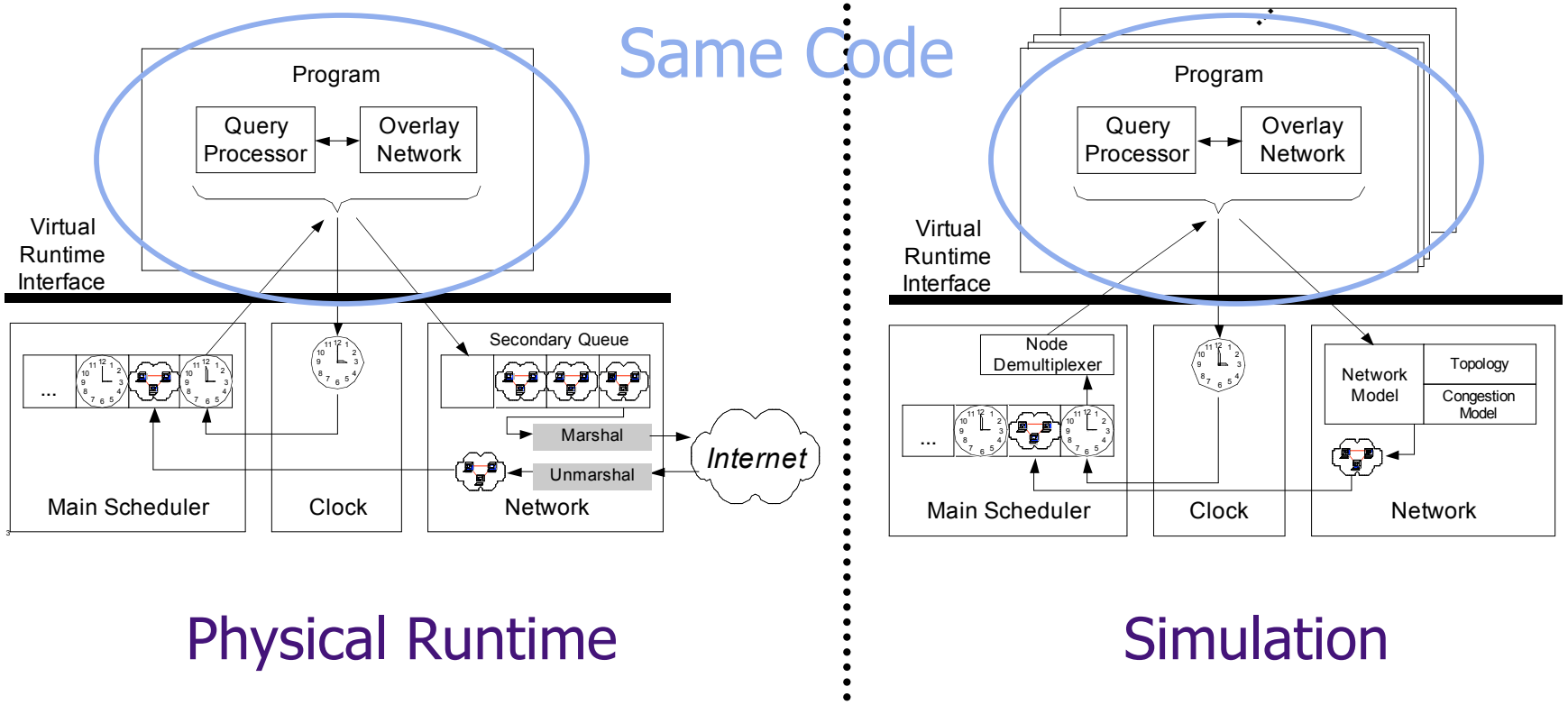


Low-level Execution Environment

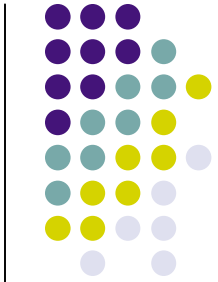


- PIER architecture implemented in Java
- Key Idea: Both simulation/physical environments use same code!
- PIER also uses events not threads, single-threaded
  - Nice for efficiency, asynchronous I/O
  - Fits naturally with discrete-event network simulator
- Uses Virtual Runtime Interface (VRI) consisting of:
  - System clock
  - Event scheduler
  - UDP/TCP network calls
  - Local storage
- At *runtime* bind the VRI to either the simulator or the OS
- Note: Event handlers must process events quickly!

Low-Level Execution Environment

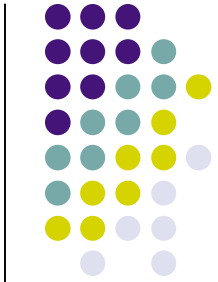


DHT Design



- DHT functionality divided into 3 components:
  - Routing Layer
    - Given *key*, maps key into *IP address* of node
  - Storage Manager
    - Provides temporary storage of DHT-based data
  - Provider
    - Ties routing layer and storage manager together
- Currently deployed with CAN & Chord.
- Note: PIER supports any DHT (i.e. CAN, Chord, Viceroy, etc)

Routing & Storage (Contd.)



- Routing Layer
  - Employs DHT API (lookup, join)
  - Callback Functions: locationMapChange → key change
- Storage Manager
  - Easily realizable API
  - Efficient performance relative to network
  - Main-memory storage manager used

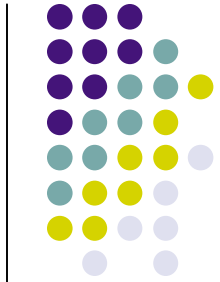
`lookup(key) → ipaddr`
`join(landmark)`
`leave()`
`locationMapChange()`

Routing Layer API

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

Storage Manager API

Provider (Contd.)

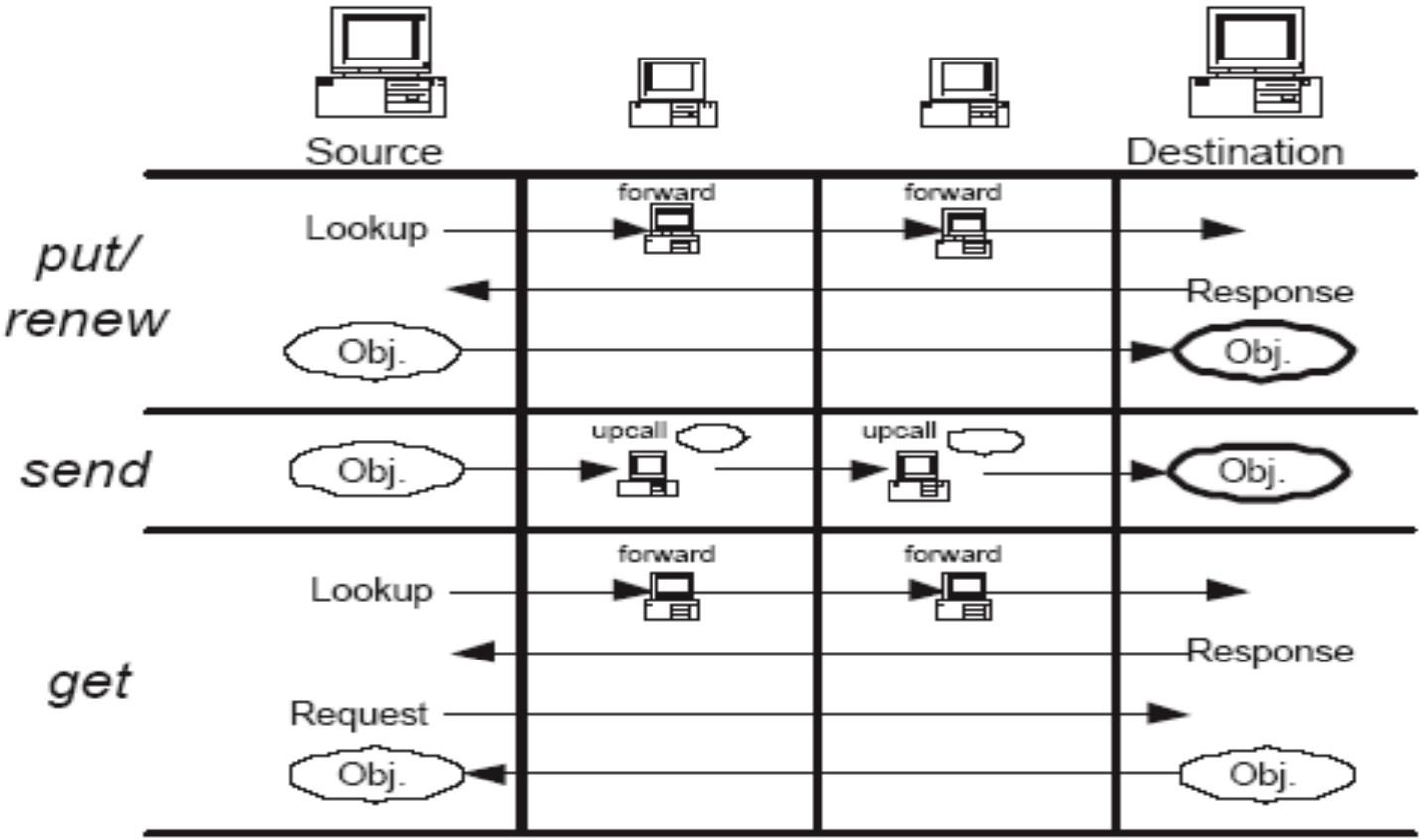


- Provides naming scheme for objects in DHT
  - namespace – relation
  - resourceId – primary key
    - namespace + resourceId → key
  - instanceId – distinguishes objects with same namespace and resourceID
  - lifetime – item storage duration *(Soft-state primitives)*
- multicast – contacts namespace's nodes
- lscan – iterates over a node's local data
- newData – application callback on data arrival

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

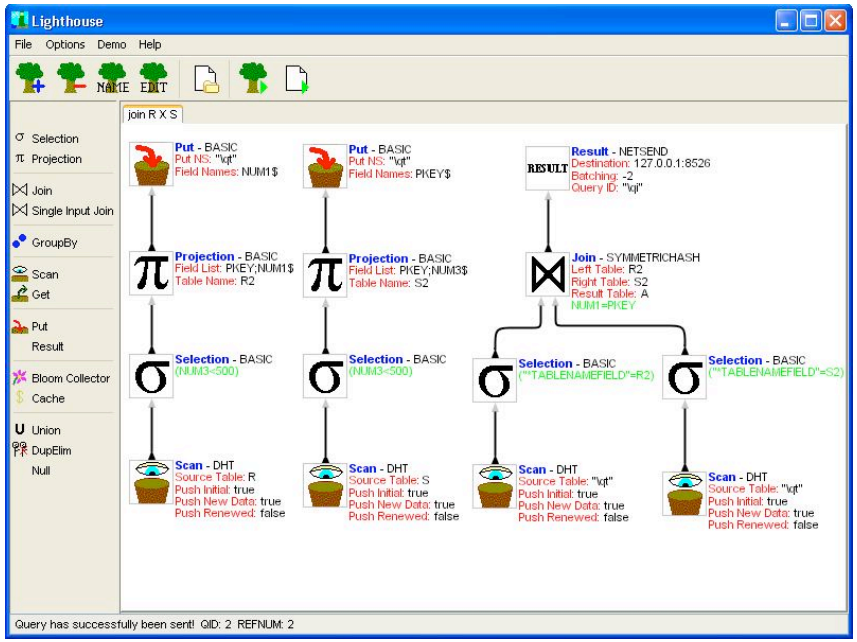
Provider API

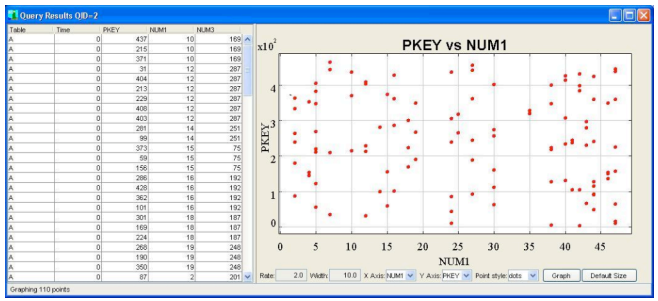
Examples of Put, Renew, Get



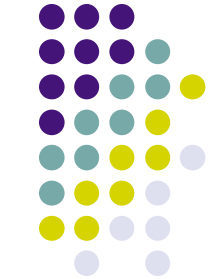
Query Processor

- Employs native algebraic data flow language called UFL (box-and-arrows graphs)
  - UFL specified physical query execution plans
  - Support operators: selection, projection, distributed joins, grouping, and aggregation
  - Operators *push* and *pull* data (using queue)
  - Updates to data via DHT interface
- GUI (called Lighthouse) for query entry and result display
  - Designed primarily for application developers
  - User inputs an optimized physical query plan



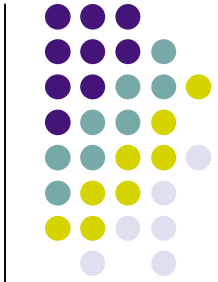


Distributed Joins



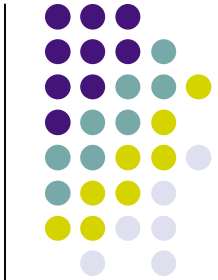
- Idea: Employ DHTs as hash table for underlying parallel join algorithms
  - Store relations (e.g. N, S) in separate namespaces
  - Use DHT-based temporary table facility for materialization of queries
- Implemented two binary equi-join algorithms
- Implemented two bandwidth reducing rewrite schemes

Distributed Joins (Contd.)



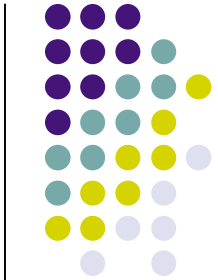
- Core Join Algorithms
  - Symmetric Hash Join (SHJ)
    - Use two *lscans* to retrieve data stored at source tables
    - Rehash eligible tuples based on join attribute (put-based)
  - Fetch Matches (FM)
    - Use single lscan to retrieve data of one source table
    - Issue *get* for matching tuples of other table (get-based)
- Join Rewriting Schemes
  - Symmetric Semi-Join
    - Run SHJ on source data projected to only hash key & join attributes.
    - Use results of join as source for two FM joins to retrieve other attributes for tuples that are likely to be in the answer set
  - Bloom Filters
    - Use of bloom filters to reduce amount of data rehashed in the SHJ
- Note: Tradeoff bandwidth (extra rehashing) for latency (time to exchange filters)

Evaluation Methodology

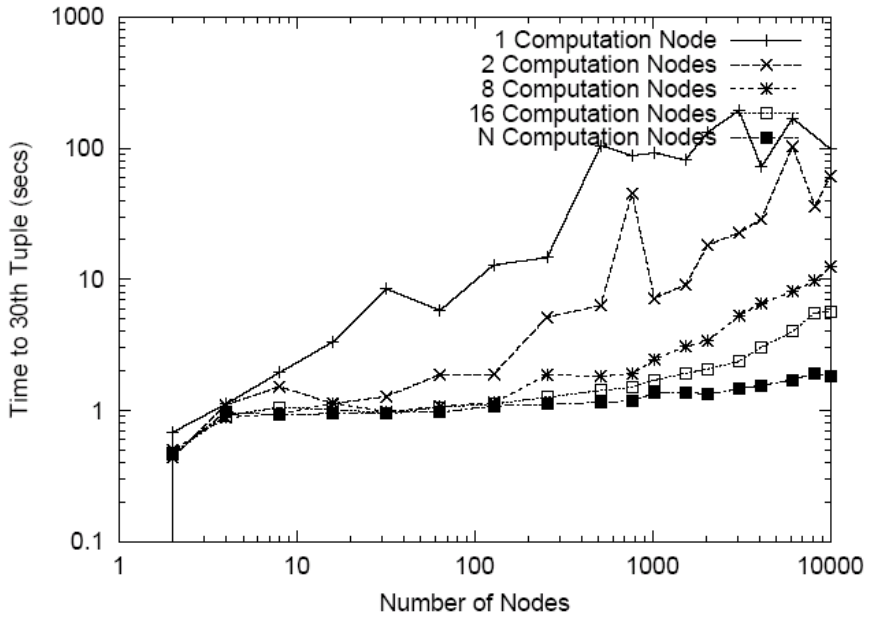


- Evaluation Metrics:
  - Maximum inbound traffic at node
  - Overall/Aggregate traffic in system
  - Time-to-receive last or 30th result tuple
- Workload:
  - *Synthetically* generated query to exercise engine
  - Use symmetric hash-join for simulations
- Network Topologies
  - Simulation Setup
    - Fully-connected network (upto 10,000 nodes)
    - GT-ITM transit-stub topology (more realistic networks)
  - Experimental Setup
    - Run on 64 PC shared cluster connected by 1-GBps network

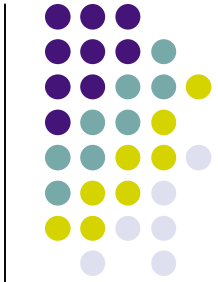
Scalability Results



- 1MB data per node
- Fully-connected topology
- Variable number of computation nodes
- Network congestion is an issue with few computation nodes
- How is the computation workload distributed?



Join Algorithms: Latency



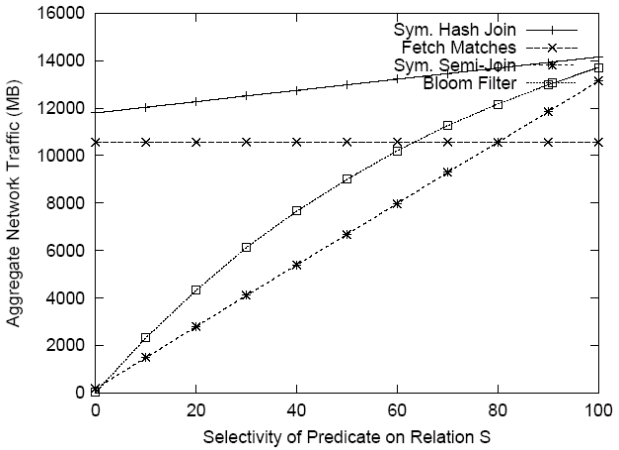
- Infinite Bandwidth
- 1024 data and computation nodes
- Core join Algorithms
  - Perform faster
- Rewrites
  - Bloom Filter: two multicasts
  - Semi-join: two CAN lookups

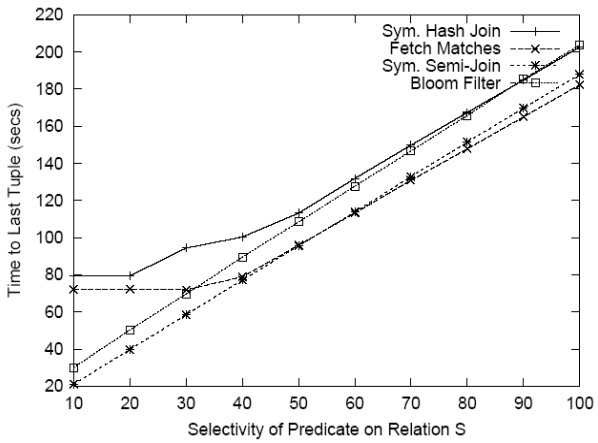
symmetric hash	Fetch Matches	symmetric semi-join	Bloom Filter
3.73 sec	3.78 sec	4.47 sec	6.85 sec

Average time to receive the last result tuple.

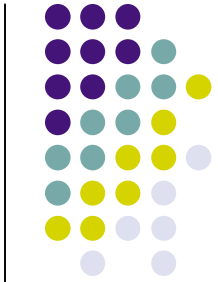
Join Algorithms: Network Capacity

- Limited Bandwidth
  - 10Mbps inbound capacity
  - 25GB relations, 1024 nodes
- Symmetric Hash Join
  - Rehashes both tables
- Semi-join
  - Transfers only matching tuples
- At 40% selectivity, bottleneck switches from computation nodes to query site

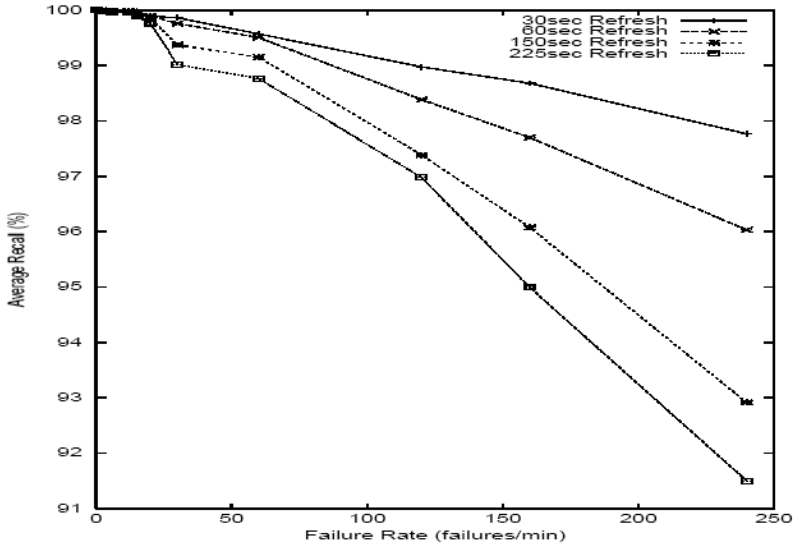




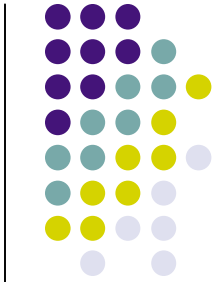
Soft State Results



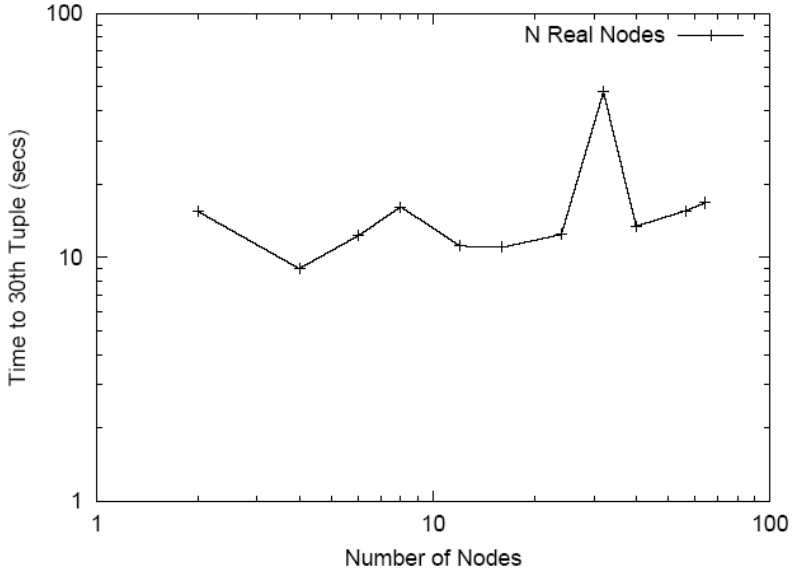
- Failure detection and recovery
  - 15 second failure detection
  - 4096 nodes
- Refresh period
  - Time to reinsert lost tuples



Experimental Results



- 64 PCs on 1Gbps network
- All nodes are computation nodes

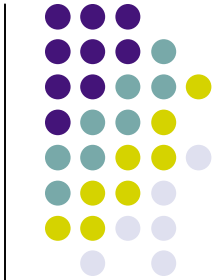


PIER Status

- Running 24x7 on 400+ PlanetLab nodes (Global test bed on 5 continents)

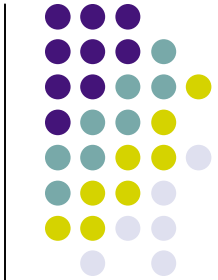
- Demo application of network security monitoring
- Extended Gnutella implementation

Future Work



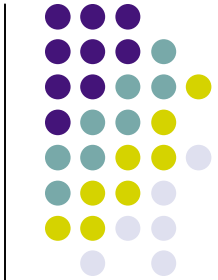
- Continuing Research
  - Optimization
    - Static optimization vs. Distributed eddies
    - Multi-Query optimization
  - Security
    - Result fidelity
    - Accountability
    - Politics and Privacy

Outstanding Issues



- Performance Concerns
  - Implementation issues
    - Network too slow
      - C program can get 340Mbps between nodes; Java only gets 100 Mbps
    - Memory Management issues
      - Object creation/garbage collection overheads
- Application Concerns
  - Requires each instance of DHT/PIER to run on each participating node
    - No way to submit queries externally to system
    - Is this the correct approach?
- Resource Management Concerns
  - Query processing hotspots
    - Realistic distributions of join attributes cause hotspots in all dimensions
  - No fine-grained monitoring and control of query executions
    - Limited control on query execution
    - No mechanisms to monitor query execution

References



[1] Ryan Huebsch, Brent Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The Architecture of PIER: an Internet-Scale Query Processor. CIDR, January 2005

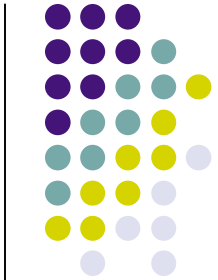
[2] Boon Thau Loo, Ryan Huebsch, Ion Stoica, and Joseph M. Hellerstein. The Case for a Hybrid P2P Search Infrastructure. *In Proceedings of the 3nd International Workshop on Peer-to-Peer Systems (IPTPS '04)*, February 2004

[3] 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 Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002

[4] Boon Thau Loo, Joseph M. Hellerstein, Ryan Huebsch, Scott Shenker, and Ion Stoica. Enhancing P2P File-Sharing with an Internet-Scale Query Processor. VLDB, September 2004

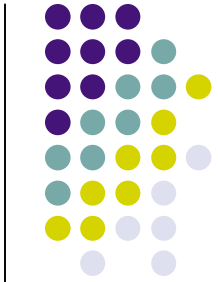
[5] Ryan Huebsch, http://www.huebsch.org/talks/CIDR05.ppt

Discussion

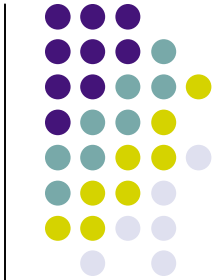


- Performance is major a problem for the PIER system
  - Many aspects of PIER can become bottleneck *(Query processing hotspots)*
  - How do we support running continuous queries?
- *Standardized* schemas don't seem to be that flexible
  - There may be many popular applications with different schemas
  - Can we establish a few "standard" schemas for all kinds of data?
- PIER currently only supports DHT-based modification/access of data
  - Is this flexible enough for all kinds of query operations?
  - What are the performance implications of doing this?
- PIER doesn't support flexible control and monitoring of queries (like ObjectGlobe)
  - Is this type of functionality important in this domain?
  - What applications would benefit from this approach?
- PIER requires *full* participation by end-user for submitting queries
  - Is this the right way to interact with the system?
  - Will this approach work for file-sharing applications?

Backup Slides…



Interesting Applications



- i3-style Network services
  - Mobility and Multicast
    - Sender is a publisher
    - Receiver(s) issue a continuous query looking for new data
  - Service Composition
    - Services issue a continuous query for data looking to be processed
    - After processing data, they publish it back into the network
- More complex services
  - Composition of e-service providers, e.g. via SOAP
  - Dataflow programs in the wide area