PIER: Internet Scale P2P Query Processing with Distributed Hash Tables

Ryan Jay Huebsch

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-52
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-52.html

May 16, 2008

PIER: Internet Scale P2P Query Processing with Distributed Hash Tables

by

Ryan Jay Huebsch

B.S. (Rensselaer Polytechnic Institute) 2001
M.S. (University of California at Berkeley) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Joseph M. Hellerstein, Chair
Professor Ion Stoica
Professor Michael Franklin
Professor Ray Larson

Spring 2008

The dissertation of Ryan Jay Huebsch is approved:

Chair Date

Date

Date

Date

University of California, Berkeley

Spring 2008

PIER: Internet Scale P2P Query Processing with Distributed Hash Tables

Abstract

PIER: Internet Scale P2P Query Processing with Distributed Hash Tables

by

Ryan Jay Huebsch
Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein, Chair

Distributed database systems have long been a topic of interest in the database research community. Existing designs focus on two principles; make the distribution *transparent* to users and provide a rich declarative query language with strict semantic guarantees. As a result, they have modest targets for network scalability with none of these systems being deployed on much more than a handful of distributed sites.

The Internet community has recently become interested in distributed query processing. 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). To achieve this degrees of scale, these system sacrifice transparency and/or flexibility.

This thesis develops a system called PIER (which stands for "Peer-to-Peer Information Exchange and Retrieval") which provides a rich query language that provides location transparency and scalability with relaxed semantics. We explore the architecture of PIER, develop techniques for query processing (with specific focus on aggregation and join operations), and finally examine an optimization problem with multiple simultaneous aggregation queries.

Professor Joseph M. Hellerstein
Dissertation Committee Chair

Contents

List of Figures

List of Tables

Chapter 1

Introduction

At any instant of time there are hundreds of millions of computers connected to the Internet. Normally just a small fraction of these machines function as servers and provide methods for sharing data, while most of the computers do not expose methods for efficiently sharing their information. This shortcoming limits the types of applications that are developed. Designers that want to utilize the huge volume of information locked inside the end-hosts are often forced to design centralized systems, which are not always suitable.

PIER (which stands for "Peer-to-Peer Information Exchange and Retrieval") was conceived to be a framework for applications desiring a pure distributed architecture. PIER enables computation where end-hosts supply the raw information and perform the calculation in a completely distributed manner with no centralized coordination. PIER provides a flexible and familiar platform for application builders using a query language composed of relational database-style operations such as aggregation and joins, along with more general dataflow operators to support a wide range of applications. The query language supports snapshot and continuous query semantics along with support for recursive queries.

PIER presents a "technology push" toward viable, massively distributed query processing at a significantly larger scale than previously demonstrated. In addition, we present an important, viable "application pull" for massive distribution: the querying of Internet-based data *in situ*, without the need for database design, maintenance, or integration. We believe in the need and feasibility of such technology in arenas like network monitoring.

In this thesis we present our design decisions within the design space, including our choice to use relatively new overlay network algorithms. We describe possible applications that could be built using PIER, such as a network monitoring tool. The majority of the thesis is devoted to describing the architecture and implementation in detail. Throughout the discussion of algorithms we present experimental results from detailed simulations.

The primary contribution of this work is to show that the use of an overlay network, in particular a class of algorithms commonly referred to as *distributed hash tables* or DHTs, are an elegant and efficient tool to enabling scaling the number of participating nodes beyond existing *parallel* or *distributed database*

systems. We show and compare methods for implementing relational joins and aggregation. Overlay networks and DHTs will be described in Chapter 2 and the use of DHTs in PIER is described in depth in Chapters 3 and 4.

The design of PIER was guided by a number of design principles discussed next. Afterwards we briefly describe potential applications that helped guide our work.

1.1 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 [35]. PIER adopts a relational data model in which data values are fundamentally independent of their physical location in 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 [56] and LDAP [37], filesharing systems like Gnutella and KaZaA, and research systems like Astrolabe [78] and IrisNet [24] – all of which use hierarchical networking schemes to achieve scalability. Analogous to the early days of relational databases, 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.

1.1.1 Network Scalability, Resilience and Performance

Traditionally, database scalability is measured in terms of database sizes. In the Internet context, it is also important to take into account the network characteristics and the number of nodes in the system. PIER achieves scalability by using *distributed hash table* (DHT) technology (see [41, 63, 66, 68, 74, 87] for a few representative references). As we discuss in more detail in Sections 2.3 and 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. DHTs are extremely scalable, typically incurring per-operation overheads that grow only logarithmically with the number of participating nodes 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 nodes in the network.

Like most Internet applications, we want our system's scalability to grow organically with the degree of deployment; this degree will vary over time, and differ across applications of the underlying technology. This means that we must avoid architectures that require *a priori* allocation of a data center, and financial plans to equip and staff such a facility. The need for organic scaling is where we intersect with the current enthusiasm for P2P systems (such as the DHT works cited previously). P2P systems gain more capacity as more participants join and contribute.

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 Chapter 4, PIER minimizes network bandwidth consumption via fairly traditional bandwidth-reducing algorithms (e.g., Bloom joins [53], multi-phase aggregation techniques [71], etc) and new optimization algorithms. 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.

1.1.2 Relaxed Consistency

While transactional consistency is a cornerstone of database functionality, conventional wisdom states that ACID transactions severely limit the scalability and availability of distributed databases [28]. ACID transactions are certainly not used in any massively distributed systems on the Internet today. Brewer neatly captures the issue in his "CAP Conjecture" [25] which states that a distributed data system can enjoy only two out of three of the following properties: Consistency, Availability, and tolerance of network Partitions. He notes that distributed databases always choose "C", and sacrifice "A" in the face of "P". By contrast, we want our system to become part of the "integral fabric" of the Internet – thus it must be highly available, and work on whatever subset of the network is reachable. In the absence of transactional consistency, we will have to provide best-effort results, and measure them using looser notions of correctness, e.g., precision and recall.

1.1.3 Decoupled Storage

A key decision we made 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 databases like BerkeleyDB, JDBC-enabled databases), to proposed Internet-scale massively distributed storage systems [19, 45].

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 the system, such as query optimization and verifying query syntax (Section 3.3).

1.1.4 Standard Schemas via Grassroots Software

An additional challenge to the use of databases – or even structured data wrappers – is the need for thousands of users to design *and integrate* their disparate schemas. These are daunting semantic problems,

and could easily prevent average users from adopting database technology. Fortunately, there is a quite natural pathway for structured queries to "infect" Internet technology: the information produced by popular software. Local network monitoring tools like Snort [67], TBIT [1] and even tcpdump provide ready-made "schemas", and – by nature of being relatively widespread – are de facto standards. Moreover, thousands or millions of users deploy copies of the same application and server software packages, and one might expect that such software will become increasingly open about reporting its properties. The ability to stitch local analysis tools and reporting mechanisms into a shared global monitoring facility is both semantically feasible and extremely desirable.

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

1.1.5 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 machines, 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).

1.2 Application Space

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: file sharing and network monitoring.

1.2.1 P2P File Sharing

File sharing was one of the first popular P2P applications in global deployment and therefore 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

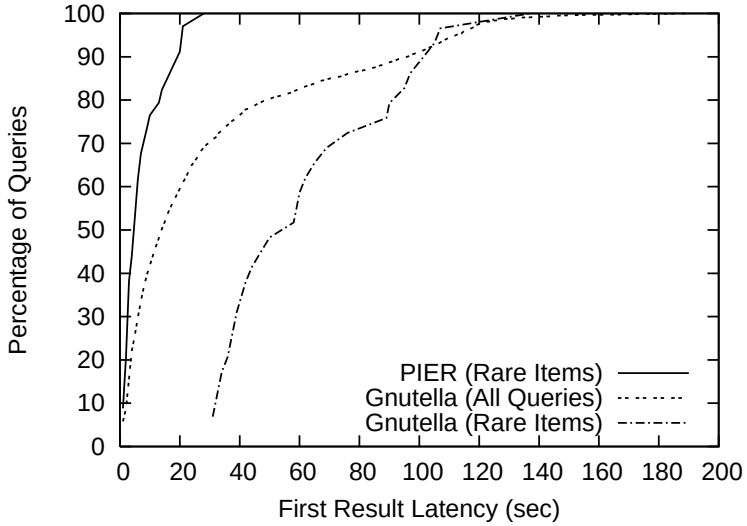


Figure 1.1: CDF of latency for receipt of an answer from PIER and Gnutella, over real user queries intercepted from the Gnutella network. PIER was measured on 50 PlanetLab nodes worldwide, over a challenging subset of the queries: those that used "rare" keywords used infrequently in the past. As a baseline, the CDF for Gnutella is plotted both for the rare-query subset, and for the complete query workload (both popular and rare queries). Details appear in [49].

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 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 [49], we deployed this hybrid infrastructure on 50 nodes worldwide in the PlanetLab testbed [62], 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 1.1 presents a representative performance graph from that study showing significant decreases in latency.

1.2.2 Endpoint Network Monitoring

The Internet today is viewed by many as a black box. Packets of data originate from one host and hopefully arrive at their destination shortly thereafter. However, when data is not flowing as expected, even experienced users are often left clueless as to why communication is failing. Furthermore, since the Internet, by definition, is the federation of thousands of smaller networks and a few large networks, there is no single authority who can provide explanations for problems. There is no single entity who has access to every parameter/setting, current status, or more broadly the global state of the Internet.

We believe it is possible to analyze partial network state from many endpoints and form a more

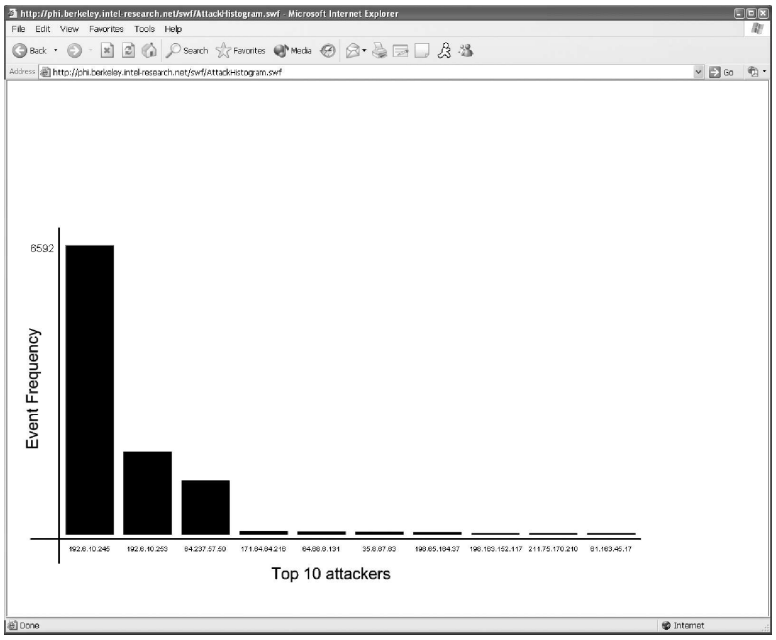


Figure 1.2: The top 10 sources of firewall log events as reported by 350 PlanetLab nodes running on 5 continents.

complete picture of the current overall state. As more hosts contribute, the picture becomes more complete and accurate. Given that network status information is already decentralized both in location and administration, a decentralized query engine would be desirable.

Specifically, 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 query answers are sufficient, and online aggregation [36] is desirable.

Figure 1.2 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 [85]. This PIER query illustrates the same result in real time, and could be used to automatically parametrize packet filters in a firewall.

Chapter 2

Background

PIER builds on a number of classic and modern topics in both database and network research. This chapter provides an overview of a number of these background concepts including distributed and parallel databases, parallel query processing methods, and overlay networks.

2.1 Distributed and Parallel Databases

Distributed databases and parallel databases are two related areas of data management work. In both cases one of the primary design goals is for the system to logically appear to the user as a centralized system. A user submits a query without knowledge of where the data is located. The database system provides the complete answer using data present throughout the entire network of participating nodes as if all the data was stored on a single node. This is a natural extension of the disk-oriented data independence feature of single node database systems. As with a centralized database system, the distributed/parallel database will optimize the query to find an efficient plan of accessing the required data and provide the same transactional support (ACID) found in a centralized system.

The distinction between a distributed and parallel database is often based on the level of autonomy and type of network connecting the participating nodes. Distributed databases are loosely coordinated autonomous systems (possibly with different administrators) often connected by wide-area networks. Parallel databases are tightly coupled nodes under the same administrative control connected by a high-speed local-area network. In a parallel database there is often a coordinator node, where all queries are submitted, optimized, and then distributed to the computation nodes for processing. However, with a distributed database queries can submitted to any node, which acts as the coordinator for that query and has the role of computation node for other queries. Distributed systems can often answer some queries when disconnected from the network if all the required data is available from the connected nodes.

In both distributed and parallel databases the data is fragmented among the nodes. For a given set of relations in a database, there are three means of fragmenting:

- **Relations:** Each node participating in the system is assigned a subset of the relations. The entire relation (all tuples and all attributes) is stored on the assigned node. For example, node 1 may store relations R and S and node 2 stores relations T and U.

- **Vertical Partitioning:** A relation is split into one or more partitions, such that each partition contains a subset of attributes (possibly overlapping) for each tuple in the relation. Each partition at a minimum contains the primary key attribute(s). This enables an equi-join on the primary key attribute(s) over all partitions to form the complete original relation. For example, relation R with primary key R_A can be partitioned in three partitions, $\{R_A, R_B, R_C\}$, $\{R_A, R_D, R_E\}$, and $\{R_A, R_B, R_E\}$.

- **Horizontal Partitioning:** A relation is split into one or more partitions, such that each partition contains a subset of the tuples. This method is sometimes called declustering. Each tuple in a partition is complete with all attributes. The complete original relation can be formed by taking a union of each partition. Horizontal partitioning can be achieved using a number of methods:

  – **Round-Robin** Tuples are evenly distributed among the partitions. Tuples are effectively randomly distributed. This method supports full relation scans efficiently, but is not appropriate for index or range queries since tuples are not stored based on value. Locating specific tuples based on value requires examining each and every tuple.

  – **Hashing** Tuples are hashed on one or more attributes (for example the primary key). Given a sufficiently good hash function, tuples are evenly distributed. This method works well for full relation scans and index scans on the hashing attributes, but is not appropriate for range queries since tuples are not ordered.

  – **Range** Tuples are divided into groups defined by non-overlapping ranges of one or more attributes. While this method works well for queries with predicates over the range attribute(s), if the ranges are not well chosen then the workload may not be equally distributed among the participating nodes.

  – **Arbitrary Predicates** A generalization of the range partitioning method, tuples are divided into groups based an arbitrary predicates (as opposed to just range predicates). To be a correct partitioning, each tuple must match at least one predicate. If tuples are allowed to match multiple predicates the tuple is effectively replicated and special care must to be taken to always update/delete all copies of the tuple. This method works well when the partitioning predicates are commonly found in the queries. However, this method is also susceptible to uneven data distribution.

Figure 2.1 illustrates vertical and horizontal partitioning. A combination of multiple of fragmentation techniques can be used simultaneously. Each fragment must be allocated to one or more nodes. Allocating the same fragment to more than one node is replication. Furthermore, replication can also occur if the fragments are overlapping. The decision on which fragmentation methods to use and an allocation strategy is based on the data and query workload. For example, if only some attributes for a particular relation are

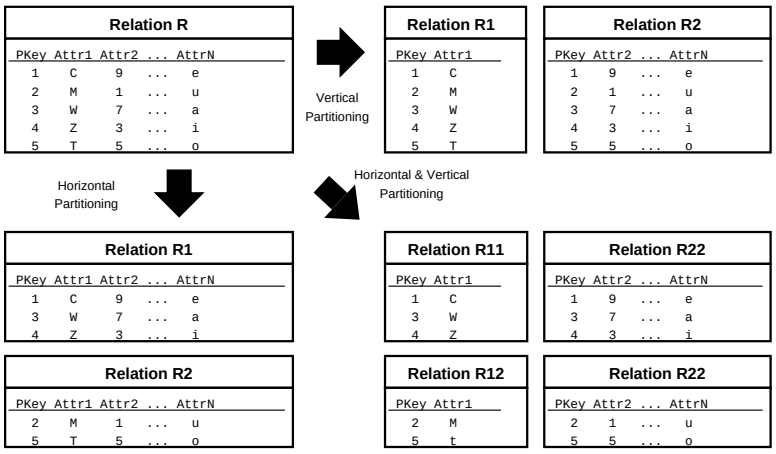


Figure 2.1: Relation R (top left) can be partitioned horizontally (bottom left), partitioned vertically (top right) or partitioned both horizontally and vertically (bottom right).

queried at one site, then a vertical partition containing those attributes can be allocated to the node at that site. Traditionally a highly skilled database administrator chooses the allocation scheme.

2.2 Distributed Query Processing Methods

There are two primary forms of distributed processing in database systems: pipelined and partitioned. Pipeline parallelism divides a query plan into blocks of operators. Each block is then assigned to a node for processing. The blocks are connected via special operators that transfer tuples from one node to another. On the other hand, partitioned parallelism is achieved by dividing the data into disjoint sets using horizontal partitioning (see Section 2.1). Each node executes the entire query plan over a subset of the data. At various points in the query plan, usually before a join or aggregation, special operators may move data between nodes such that complete join or aggregation buckets/groups are placed on the same node.

The majority of the literature for distributed query processing focuses on methods for joins. The early systems considered two methods: shipping entire relations and "fetching " tuples for index joins. These methods are best suited to small networks where there are few participating nodes. For example, consider a join where the one relation, R, is stored at node 1 and the other relation, S, is stored at node 2. Two possible query plans include having node 1 send a copy of relation R to node 2 where the join is performed, or vice-versa having node 2 send a copy of relation S to node 1. Figure 2.2 illustrates the first scenario.

The Fetch Matches method [53] is a specialization of the relation shipping method. Instead of sending the entire relation from one node to another, the node with the R relation (also called the "outer relation") sends a (fetch) request for some tuples to the node with the S relation (also called the "inner" relation). The requested tuples are then forwarded to the requesting node. Unlike the shipping the entire table, if there are tuples in the S relation that are not needed for the join they will not be sent across the network. Figure 2.3 illustrates an example fetch matches query.

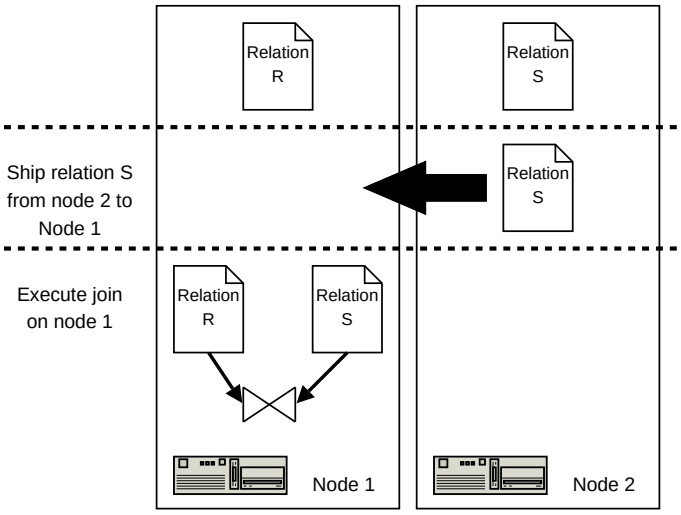


Figure 2.2: Relation R is stored on node 1 and relation S is stored on node 2. To compute the join node 2 sends a copy of S to node 1.

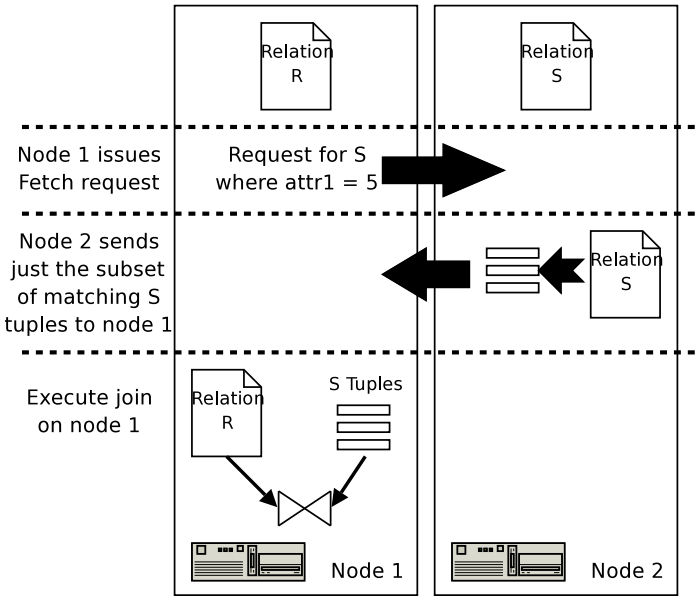


Figure 2.3: Fetch Matches: Relation R is stored on node 1 and relation S is stored on node 2. After scanning R, node 1 requests S tuples where attribute c is equal to 'blue'. Node 2 sends the tuples to 1 where the join is performed.

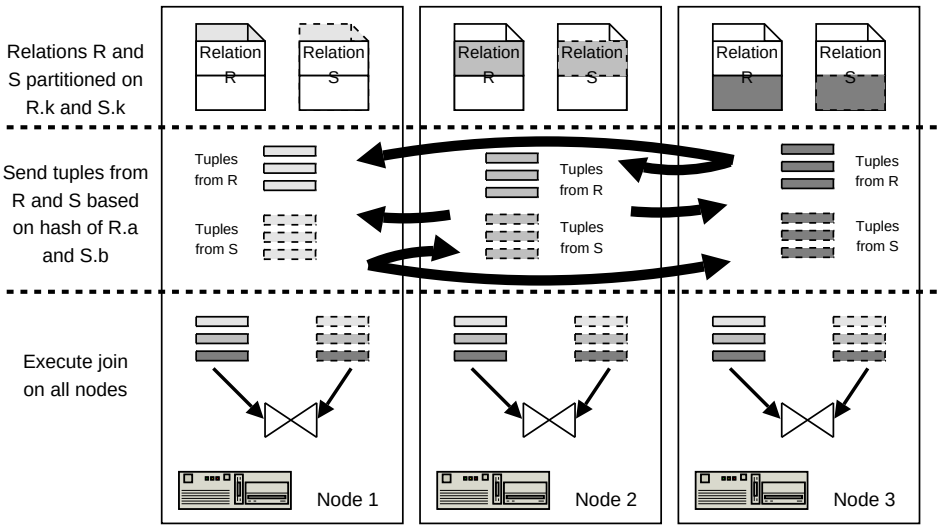


Figure 2.4: Hash Join: Relations R (and S) are horizontally partitioned by hashing attribute $R.k$ (and $S.k$). To compute the join on $R.a = S.b$, all nodes scan their partitions of R (and S) and send a copy of the tuple based to another node based on the hash of $R.a$ (and $S.b$).

Both of these methods were designed for systems where entire relations are stored at individual nodes as opposed to using horizontal partitions. These methods work well with pipelined parallelism.

Parallel query processing systems (such as the Gamma system [20]) build on the relation-shipping concept. Since parallel systems primarily use horizontal partitioning, they dynamically create and move fragments instead of entire relations to execute joins and aggregations. A parallel hash join is executed by redistributing tuples among the nodes using a hash function on the join attribute(s). Each hash bucket is assigned to a node using a mapping function known by all nodes. Once all the source tuples are hashed into their proper buckets, each node can compute the join or aggregation over the hash bucket locally. A three-node example is shown in Figure 2.4. This is a straightforward extension of the centralized hash join, with the exception that individual hash buckets maybe located on different nodes. The use of hashing instead of range or arbitrary predicates was chosen to help achieve an even distribution of tuples and work.

2.3 Overlay Networks

A network can be broadly defined as the collection of communication links between a set of nodes, along with the addressing and routing scheme used. A small Ethernet network may contain two or more nodes connected to a switch. Each node on this network has a network interface card with a built-in Ethernet address (also called the MAC address). The routing is handled by the switch which delivers each packet of data to appropriate node based on the Ethernet address.

An overlay network is network that is built on top of another network. It can provide new addressing and routing protocols with the goal of providing a new service not achieved using the lower network(s). In

essence, overlay networks are a means of inserting a layer of indirection above the network.

The Internet (and the IP protocol) can be considered an overlay network. The Internet allows for communication across all participating networks. Nodes in the Internet are assigned addresses and use a variety of routing protocols. However, these addresses and protocols work on top of the native schemes used by the underlying network.

Overlay networks can also be used to provide content-based routing. In content-based routing the source specifies the destination based on the value of the data it is sending or retrieving. The destination node is then determined on the fly by the overlay network. Over time as nodes join and leave the system the destination node for a particular value may change. The IP protocol requires the source to specify the destination node directly, thus forcing the sender to determine the exact destination node.

When evaluating different content-based routing algorithms the key metrics include latency, consistency and robustness. Many implementations use multi-hop routing, which can be much slower than direct routing, increasing the latency of a message. Reducing the number of hops or reducing the latency of each hop can improve the overall latency. Consistency is the measure of whether different messages (from the same or different source nodes) are routed to the same destination node. Many content-based routing algorithms claim "eventual consistency" which means that if the network remains stable for some period of time the routing becomes completely consistent. Finally robustness measures the system's ability to remain (mostly) consistent despite nodes joining, leaving and network interruptions.

There have been a large number of content-based routing algorithms introduced. Our work primarily used one, Bamboo [66] which we briefly describe below.

2.3.1 Bamboo

Bamboo is a content-based overlay router loosely based on Pastry [68]. Bamboo was engineered to be extremely robust to node failures and network disruptions. In particular the design of Bamboo is based on periodic (instead of reactive) recovery from failures, careful calculation of message timeouts, and uses proximity neighbor selection.

Bamboo uses a 160-bit flat identifier space grouped into x digits of $log_2 b$ bits where $x = \frac{160}{log_2 b}$. The parameter b is commonly referred to as the base, with 16 (hex digits) as a common value. The space can be visualized as circle where the identifier wraps around at 2^{160}. Figure 2.5 shows an example Bamboo network.

Each node is randomly assigned an identifier (sometimes the hash of the node's IP address is used). Since the space of identifiers is much larger than the number of nodes, there are many identifiers that do not map directly to a node. Instead, identifiers map to the node numerically closest (with wraparound at 0 and 2^{160}. Each node maintains two sets of neighbors or links: a *routing table* and a *leaf set*.

The leaf set reliably maintains a set of links to other nodes that are "nearby" in the circle, the l nodes that immediately precede and follow the node in the identifier space. The leaf set may only contain active/live nodes, since the leaf set is required for correct routing.

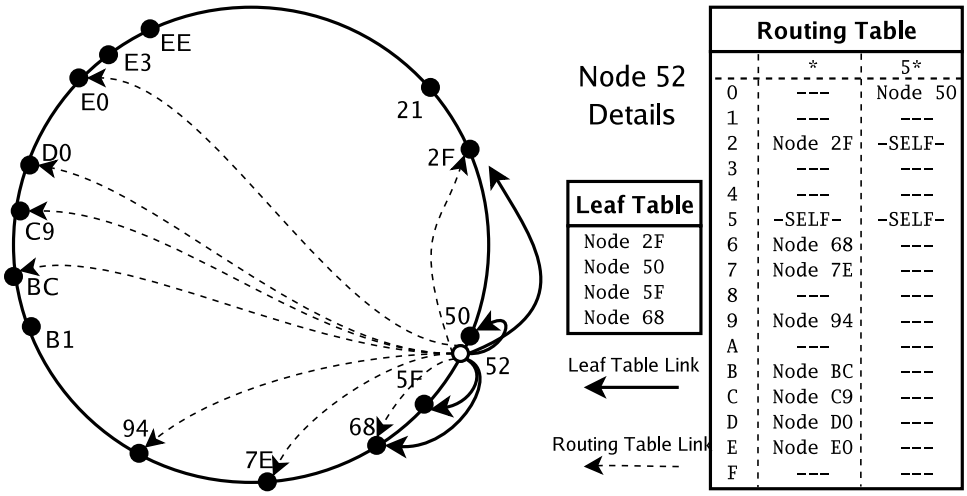


Figure 2.5: An example Bamboo network with a 8-bit, base 16 identifier space. Leaf (solid lines) and routing (dashed lines) tables/links for Node 52 are shown.

In addition to the leaf set, there is also a routing table. For each prefix of the node's identifier, the routing table contains links for b nodes such that each of those b nodes have the same prefix and different digits for the digit following the prefix. The routing table is used to reduce the number of hops when routing a message. Bamboo is less aggressive in maintaining only active nodes in the routing table than in the leaf set since it is only used for efficiency. This means Bamboo may check the node for liveliness less frequently than a leaf set node, saving resources.

The combination of the leaf set with "short distance" links (in the identifier space) and the routing table with "long distance" links allows Bamboo to efficiently, reliably and consistently route to any identifier. In the worst case, the leaf set is sufficient (although not efficient) for routing a message. Further details about Bamboo are beyond the scope of this dissertation and can be found in [66].

Chapter 3

Architecture

In this chapter we discuss the architecture of PIER. Figure 3.1 shows an overview of the architecture. The architecture is composed of three main parts: the execution environment (not shown in the figure), the distributed hash table (DHT), and the query processor. The query processor coordinates the execution of local dataflows on all participating nodes while the DHT routes data between nodes. The execution environment provides a event-based style of multiprogramming to enable easy simulation and deployment.

In the following sections we discuss each of the main components starting from the foundation and working our way up to the query processor.

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. This permits the query processor to issue many simultaneous requests for data without waiting for a request to complete before issuing the next request. This can significantly reduce latency and this strategy is employed in all query processors. It also needs to support native simulation (Section 1.1.5). 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 layer 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 long running computation tasks, to network protocols, and to the internal PIER scheduler that dispatches timer, computation, and network events. A representative set of the methods provided by the VRI are shown in Table 3.1.

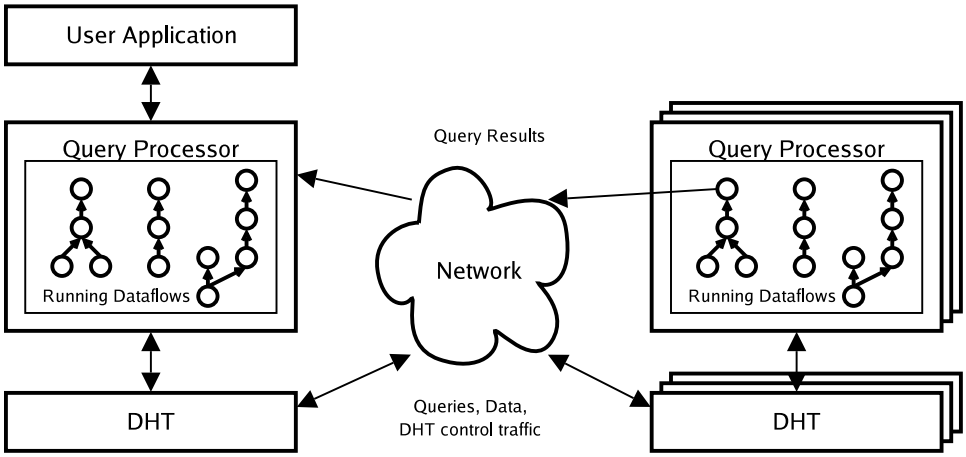


Figure 3.1: The general architecture of PIER. A query is submitted to query processor by a user application on any node. The query is executed on multiple nodes using the DHT as the primary method communication. Query results can be sent directly to the user application over the network without going through the DHT.

Clock and Timer Scheduler

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

Computation Scheduler

void $scheduleComputation$(data, callbackClient, computationFunction)
void $handleComputation$(data, result)

UDP Network Protocol

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

TCP Network Protocol

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 3.1: Selected methods in the VRI.

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. Most events in PIER are processed by a single thread with no preemption. A special class of events, *computation events*, are processed in separate threads.

The single-threaded, event-based approach has a number of benefits for our purposes. Most importantly, it easily supports our goal of native simulation. Discrete-event simulation is the standard way to simulate multiple networked machines on a single node [60]. 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[1]. 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[2]. Similarly, any long chunk of CPU-intensive code must yield the processor after some time, by scheduling its own continuation as a timer event, or be scheduled as computation 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.

Computation events are relatively long running tasks (longer than a few milliseconds) that do not support preemption. When the computation is complete an event is inserted into main queue to handle the result. Special handling of these tasks is important for both the runtime and simulation environments. Since these tasks do not complete relatively fast it is important to account for the running time in simulation. In runtime long running tasks on the main queue may significantly delay execution other events. As of now, only the multi-query optimization algorithms (described in Chapter 5) require the use of computation events.

All events originate with the expiration of a timer, with the completion of an I/O operation or at the completion of computation event.

[1] We do not take a stand on whether scalability in the number of threads is a fundamental limit [81] or not [79]. We simply needed to work around Java's current limitations in our own system.

[2] At the time of PIER's design Java did 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.

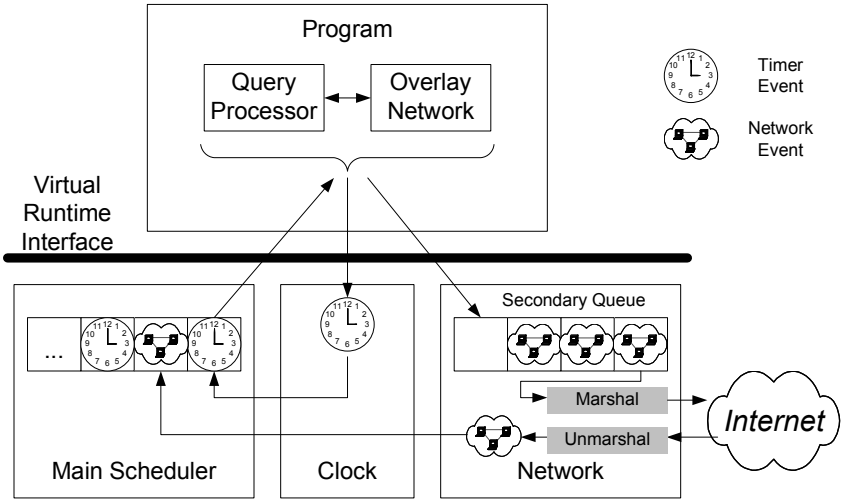


Figure 3.2: Physical Runtime Environment - A single priority queue in the Main Scheduler stores all events waiting to be handled. Events are enqueued either by setting a timer or through the arrival of a network message. Out-bound network messages are enqueued for asynchronous processing. A second I/O 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 asynchronous I/O thread, and a set of IP-based networking libraries (Figure 3.2). 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 reliable delivery or congestion control. To overcome these limitations, we utilize the UdpCC library [66], 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 and for some types of data sources. TCP facilitates compatibility with standard clients and has fewer problems passing through firewalls and NATs.

The physical runtime environment uses at least two threads. One thread for processing the main event queue and a second thread dedicated to I/O handling. A separate thread for I/O is used to ensure that the acknowledgments used in UdpCC are sent in a timely fashion. Additional threads are used for synchronous I/O data sources and for each computation event.

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 3.3). 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.

Computation events are handled like normal events, except the wall-clock time to perform the execution is recorded. After applying an optional fudge-factor multiplier to the observed execution time, the result handler for the computation is scheduled to be called in the simulator after the calculated delay. The fudge factor can be used to adjust for nodes that have slower CPU's than the machine being used to run the simulation. Additionally, the fudge factor can be used to approximate CPU sharing that is likely to occur in real environments when nodes are simultaneously used for other applications.

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 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 three standard network topology types (star, transit-stub, and the King model [32]) 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 Distributed Hash Tables (DHTs)

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 join and leave. 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 [57], also used in PeerDB [59]). 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. 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, and with high probability provides consistent routing such that at any given time any node that attempts to resolve an identifier to a node will discover the same mapping.

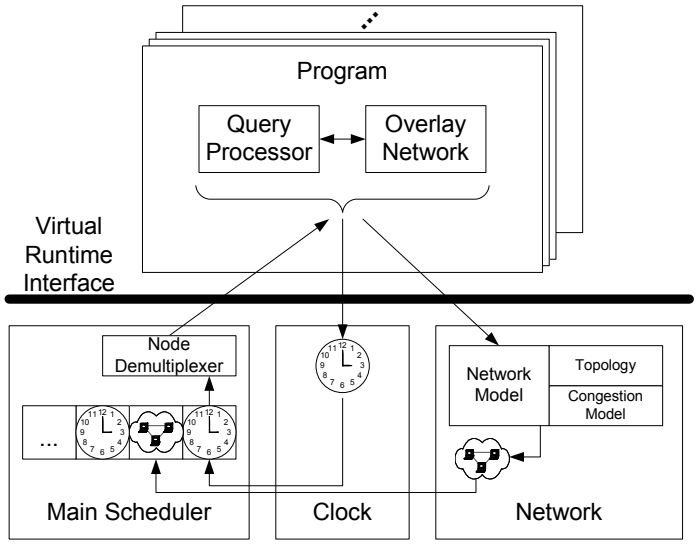


Figure 3.3: Simulation Environment - The simulator uses one Main Scheduler and priority queue for all nodes. Simulated events are annotated with virtual node identifiers that are used to demultiplex the events to appropriate instances of the Program objects. Out-bound network messages are handled by 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 out-bound message later affects the calculation.

While the core of the DHT is simply a router for the abstract identifier space, it is straightforward to include a hash-table like interface where the hash buckets are distributed throughout the network. The DHT in PIER provides a hash table's traditional get and put methods as well as additional object access and maintenance methods. For modularity, the DHT in PIER is divided into three components: the *router*, an *object manager* and a *DHT wrapper* that implements the basic API. The components are shown in Figure 3.4.

The router contains the peer to peer overlay network routing protocol (introduced in Section 2.3) of which there are many options. We currently use Bamboo [66] (described in Section 2.3.1), although PIER is agnostic to the actual algorithm, and used other DHTs including CAN [63] and Chord [74] in various stages of its development. We chose Bamboo since it is implemented in Java, the implementation was robust, and was developed locally.

In the following sections we discuss the naming of objects in the DHT, the use of soft-state for reliability, and a discussion of the API and its implementation.

3.2.1 Naming

Within the DHT every object has an *routing identifier* and a *storage name*. In many cases the storage name is used to compute the routing identifier, but it some cases they may be different. The routing identifier is used to determine which node in the network should be responsible for the object, while the

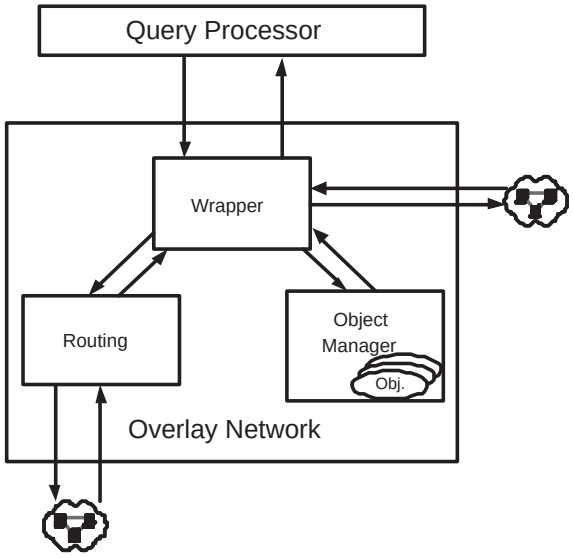


Figure 3.4: The overlay network is composed of the router, object manager and the DHT wrapper. Both the router and DHT wrapper exchange messages with other nodes via the network. The query processor only interacts with the DHT wrapper, which in turn manages the choreography between the router and object manager to fulfill the request.

storage name is used by the query processor to distinguish objects in requests.

The storage name of each object is composed of three parts: a namespace, partitioning key, and suffix. All three parts are chosen by the application that gives the object to the DHT for handling. 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. Any object with same namespace, partitioning key and suffix is considered the same object and previous instances of the object are replaced with the new object.

By default, the DHT computes an object's routing identifier using the namespace and partitioning key; the suffix is only used to differentiate objects that would otherwise share the same storage name. In our system the SHA1 hash of the namespace string concatenated with a separator (a period) and the string representation of the partitioning key is used as the default routing identifier.

For some queries, the routing identifier may be pre-specified. For example in aggregation queries, the node issuing the queries may want to be the root of the aggregation tree (details described in Chapter 4) and provides it own ID as the root routing identifier. Data sent to the root via the DHT sill has a normal storage name (namespace, partitioning key and suffix), but the storage name is not used for routing decisions.

When data is passed from the DHT to the query processor it is annotated with the storage name, but not the routing identifier. Therefore the default operation of the DHT completely masks the routing identifiers from the query processor. Only in special cases does the query processor provide a custom routing identifier to the DHT.

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 thesis, but is well-treated in [31].

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 Chapter 4.

3.2.3 Soft State

Recall that PIER 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 [14].

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 routing identifier will be mapped to a different node than before, which will not recognize the storage identifier, causing 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 it 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 and the system 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.

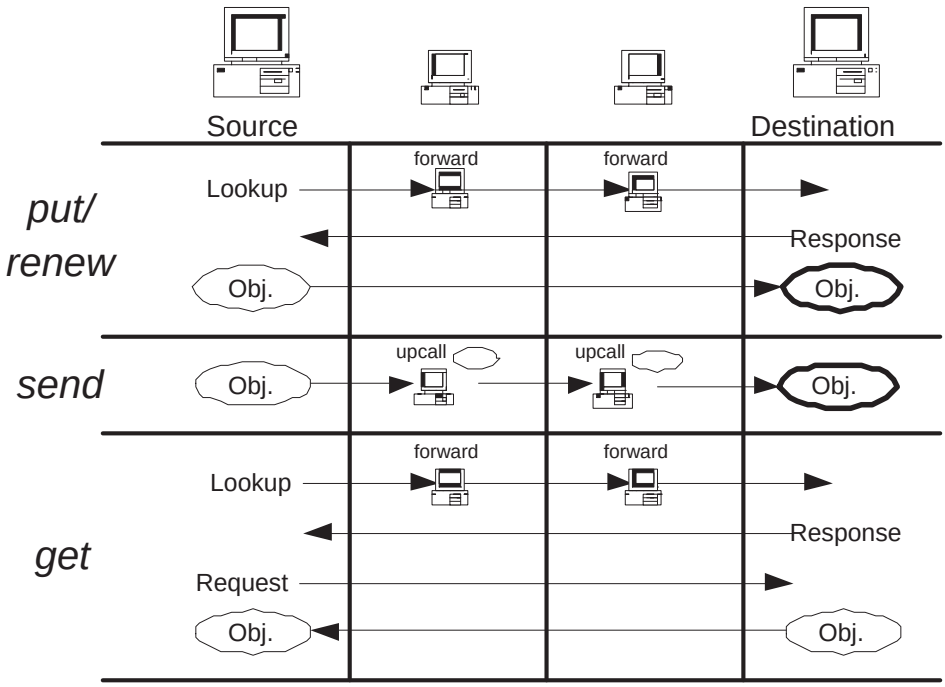


Figure 3.5: put and $renew$ perform a lookup to find the object's identifier-to-IP mapping, after which they can directly forward the object to the destination. $send$ is very similar to a put, except the object is routed to the destination in a single call. While $send$ uses fewer network messages, 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.

3.2.4 Implementation

As listed in Table 3.2 the DHT supports a collection of inter-node and intra-node operations implemented by the overlay wrapper. 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 to perform the operation. The lookup operation involves routing a small message via the overlay router. The destination node sends a direct IP response to the sender which can now send the full message to the destination node. 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 3.5 illustrates how each of the operations is performed.

The four inter-node calls each have three variants. The default variant, the application supplies just the storage identifier and the DHT computes the location identifier. The second variant allows the application to supply both the storage identifier and routing identifier. Finally, in some cases the query processor does

Inter-Node Operations

void get([locationID — remoteAddress], namespace, key, callbackClient)
void put([locationID — remoteAddress], namespace, key, suffix, object, lifetime)
void $send$([locationID — remoteAddress], namespace, key, suffix, object, lifetime)
void $renew$([locationID — remoteAddress], 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, midway, locallySent)

Table 3.2: Selected methods provided by the overlay wrapper

not want to specify a routing identifier, but instead has determined the IP address of the remote node through another means (for instance hard-coded in the query). In this last case the DHT is able to skip the lookup step and can forward the message/request directly to the remote node.

The three intra-node operations are also key to the operation of the query processor. $localScan$ and $handleLScan$ allow 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. As an optimization, all three types of requests, $localScan$, $newData$, and $upcall$ can be limited by specifying a storage identifier prefix that must be matched.

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. It is important to note that PIER is unique in its aggressive reuse of DHTs for a variety of purposes traditionally served by different components in a DBMS. These include: query dissemination (Section 3.3.3), hash index (Sections 3.3.3 and 4.3.1), range index (Section 3.3.3), partitioned parallelism, operator state (Section 3.3.5), and hierarchical operators (Sections 4.2 and 4.3).

We introduce the PIER query processor by first describing its data representation and then explaining the basic sequence of events for executing a snapshot or continuous 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 web pages, files, JDBC, 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 3.3.4). 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 accesses the value.

3.3.2 Life of a Query

For PIER we defined a native algebraic ("box and arrow") dataflow language called UFL (which stands for the "Unnamed Flow Language"). UFL is in the spirit of stream query systems like Aurora [2], and router toolkits like Click [43]. 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 specification of the language can be found in Appendix A. A graphical user interface called Lighthouse is available for more conveniently "wiring up" UFL. PIER supports UFL graphs with cycles, though such recursive queries in PIER are the topic of research beyond the scope of this thesis [50].

An UFL query plan is made up of one or more operator graphs called *opgraphs*. Each individual opgraph is a connected set of dataflow operators (the nodes) with the edges specifying dataflow between operators (Section 3.3.4). Each operator is associated with a particular implementation.

Separate opgraphs are formed wherever the query redistributes data around the network and the usual local dataflow channels of Section 3.3.4 are not used between sets of operators (similar to where a distributed Exchange operator [26] 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 dissemination (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 3.3.5), and establishes the dataflow links (Section 3.3.4) between the

operators.

During execution, any node executing an opgraph may produce an answer tuple. The tuple or batches of tuples are forwarded to the client's proxy node. The proxy then delivers the tuples 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. This is akin to the soft-state handling of objects in the DHT.

Given this overview, we now expand upon query dissemination and indexing, the operators, and dataflow between the operators.

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 [11, 64]. The method we describe here is based upon the distribution tree techniques presented in [11].

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. For example, Bamboo produces distribution trees that are have high fan-in for well connected nodes, Chord [74] produces trees that are (roughly) binomial; Koorde [41] produces trees that are (roughly) balanced binary. 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 the opgraph message to the 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. Multiple distribution trees can be used for robustness.

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[3].

PIER currently has three kinds of indexes: a broadcast-predicate index, an equality-predicate index, and a range-predicate index. The broadcast-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 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 [65]. 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 [53], 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 secondary indexes, or indexes that whose values are the keys for another index. 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. These methods are discuss further in Chapter 4.

3.3.4 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 result, 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

[3] 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; its main memory access method uses a hash tables data structure.

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. There are three types of probes: set, stream, and set-predicate. Set probes are a request for all data that is presently available at this node at this time (excluding any local I/O such as disk). Stream probes are a continuous request for new data as it becomes available at this node, existing data is not retrieved. Finally, a predicate can be embedded in a set probe that limits the data retrieved. This is a standard optimization to push-down a selection predicate into a data source and permits index lookups.

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. Queues enable recursive dataflows. Without queues, the stack depth would increase with each level in the recursion. A queue allows the stack to unwind between each successive level. Additionally since the queue yields control, starvation of other queries/opgraphs does not occur.

An arbitrary tag is assigned to each probe request, the built-in PIER operators use an incrementing counter to generate unique tags. 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).

Finally, the local dataflow allows signals to be passed from child to parent. The first signal currently supported is an end-of-probe signal that is generated when a data source has finished pushing all tuples that match a set (or predicate-set) probe.

3.3.5 Operators

PIER is equipped with 15 logical operators and 29 physical operators (some logical operators have multiple implementations). Most of the operators are similar to those in a DBMS, however PIER also uses a number of non-traditional operators, particularly for the access methods. Table 3.3 lists the operators and implementations.

Basic Operators

Operator	Implementation	Brief Description
selection	Basic	Evaluates a predicate against a tuple.
projection	Basic, Generate	Modifies the attributes of a tuple.
tee	Basic	Logically combines all parents to appear as one.
union	Basic	Logically combines all children to appear as one.
join	SymmetricHash, Index	Performs the relational equi-inner-join operation.
group-by	Basic, DualFlow	Aggregates tuples with the same values for the group-by fields.
duplication elimination	Basic	Eliminates duplicate tuples.
scan	DHT, DHTMessage, IP, JDBC, CSV	Retrieves tuples from a data source.

Other Operators

Operator	Implementation	Brief Description
put	DHT, DHTMessage, IP, Bloom, Eddy	Stores a tuple in non-query specific storage mechanisms.
result	NetSend	Sends result tuples to query requester.
flow control	Basic	Controls the timing and flow of data through an opgraph.
queue	Basic	Temporarily holds tuples during processing.
cache	MemCache, MemUpdateCache	Stores tuples during the execution of the query.
eddy	Basic	Dynamically reorders the execution of joins in query plan as described in [4].
null	Null, NullSource, NullSink	A placeholder for operators that do not match other existing operators.

Table 3.3: Selected list of PIER operators and implementations.

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. Since network throughput and latency are the dominating bottlenecks this design decision does not have a large effect on PIER's current usage model.

Physical operators can be classified into four main categories: sources, sinks, pass-through, and flow-modifying. Sources are operators that upon receiving a probe fetch or create zero or more tuples. If the source is not capable of handling a particular probe (for example it cannot evaluate a predicate-set probe) the probe is passed to its children if any exist. Depending on the type of probe and the data source, the source operator can optionally produce a EOP signal.

Sinks are operators that consume tuples removing them from the dataflow. The result operator is the most common sink operator, but others such as an operator to insert data into the DHT are also sinks.

Pass-through operators modify or drop data as it moves through the dataflow. These operators do not respond to probes and pass them through to their children. Selection and projection are examples of pass-through operators.

Flow-modifying operators may change or issue new probes during the course of processing. Examples of flow-modifying operators are joins, some group-by operator implementations and the control flow operator. For example in an index join, when the join operator receives an outer tuple it will issue a predicate-set probe for the matching tuples from the inner relation. The flow control operator issues probes based on a number of events such as the starting of a query, periodically, or based on the number or rate of tuples arriving at a data source.

Throughout the development of PIER the granularity and scope of operators have changed. In the early version of PIER monolithic operators that handled multiple aspects of the dataflow were used. For example the first join was called a DHT join and a single operator handled fetching data from the DHT, evaluating selection predicates, and joining tuples. These monolithic operators were broken up into smaller operators that are better-scoped and easily reused for multiple query plans.

3.3.6 Flow Control

Related to the evolution of the operators, was the changes in managing flow control. For example, the initial implementation of the group-by operator was designed as a pass-though operator. At the start of the query the data source would begin receiving/retrieving tuples. Sometime later when a result was desired, the flow control operator at the root of the tree would issue a set probe down the tree to the data source. The data source would then send all the collected tuples received. The group-by operator would aggregate the tuples and then send the result tuple(s) once it received the end-of-probe signal from the source.

This design suffered from two problems. First, the data source was responsible for storing the data until the probe was issued. Second, the flow control operator had limited information to decide when to issue the probe. Often a simple timer in the flow control operator was used to trigger the probe. This is often not the best method and will be discussed in depth in Section 4.2.

A later implementation of the group-by operator integrated functions of the flow control operator

directly into the group-by operator. The group-by operator, instead of the flow control operator would issue probes to the data source. While this allowed the group-by operator more flexibility in timing probes and result generation, it still required the source to store the tuples. Additionally, this solution was not elegant with two operators (the group-by and flow control operators) having similar functions and duplicated code.

The final implementation of the group-by operator, called "Dual Flow", uses two flow control operators in the query plan. One flow control operator pulls data from the source into the group-by operator, which aggregates the data as it arrives. A second flow control operator is used to retrieve the current set of aggregates/results from the group-by operator. This design allows data to be immediately aggregated when it arrives requiring no storage. The second flow control operator can issue a stream probe allowing the group-by operator to determine when to generate result tuples, or the flow control operator can control result generation and issue a set probe.

3.3.7 Error Handling

Error handling can be especially difficult given some of PIER's design decisions. The lack of system catalogs means queries can be not be statically validated against a schema. The existence of an attribute or its data type is not known until the query is executed. Additionally, the primary query interface is UFL, which unlike SQL is a low-level physical dataflow description that is unfamiliar to most users. All but the best query writers will require debugging tools (beyond a simulator) to understand what is happening inside the execution of the plan. Finally, runtime errors caused by unavailable data sources or other transient conditions can easily overwhelm a client if every node processing the query was to send an error message.

PIER provides a number of facilities to help a user deal with errors including: optional error messages, silent dropping of malformed tuples, and the ability to view a copy of tuples flowing into or out of any dataflow operator.

In a wide-area decentralized application it is likely that PIER will encounter tuples that do not match the schema expected by a query. This can be caused by a poorly written query, different versions of the data source at the node, or errors from the data source itself. 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. An optional error message can be requested which is forwarded to the user. This feature is useful to debug the initial operation of a query, but when there is a fault in a production system the number of error messages can easily overwhelm a client. By default, tuple errors are silently dropped, but it can be enabled on an operator-by-operator basis.

The input or output tuple streams to each operator can be "tapped" and forwarded to the user client. The user specifies whether to activate this feature in the query request. During the execution of the query PIER will then send a duplicate copy of each input/output tuple to the user client as if it was a result tuple. This enables the user to peek inside a running dataflow and see which tuples (and their schemas) are in which

parts of a dataflow. This feature is also enabled on an operator-by-operator basis. There is no design reason this feature could not be activated and deactivated after a query was started, however the interface to do this was never implemented.

Since error messages and internal dataflow streams are both forwarded directly to the PIER proxy node of the client, there is no opportunity to drop duplicate messages or provide a global rate-control mechanism. PIER does allow the client to specify a limit to the number of messages received from any one node, however in very large networks this has limited benefit. This leads to the default policy that all but parser errors (which originate only from the proxy node itself) are silently dropped.

3.3.8 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. This depends on the data sources (usually the DHT) to buffer the data for short time until the query is started on the local node. In our experiments tuples sent via the DHT have at least a one minute timeout which is more than sufficient.

Chapter 4

Query Processing

In this chapter we describe the main query processing algorithms, including aggregation and joins. Our goal is to develop algorithms that optimize three key metrics: minimize overall network usage, minimize answer latency, and distribute network communication equally among all participating nodes. We present experimental results alongside the algorithm descriptions. Therefore, we start this chapter with a description of the experimental setup.

4.1 Experimental Setup

Traditionally, database scalability is measured in terms of database sizes. In the Internet context, it is also important to take into account the network characteristics and the number of nodes in the system. Even when there are plenty of computation and storage resources, the performance of a system can degrade due to network latency overheads and limited network capacity. Although adding more nodes to the system increases the available resources, it can also increase latencies. The increase in latency is an artifact of the DHT algorithm we use to route data in PIER (as described in Section 3.2). In particular, with Bamboo – the DHT scheme we use in our system – a data item that is sent between two arbitrary nodes in the system will traverse $log_b n$ intermediate nodes on average, where b is a parameter of the DHT (set to 16 in our tests) and n is the number of total nodes in systems.

To illustrate these impacts on our system's performance we use a variety of metrics, including the maximum in-bound traffic at a node, the aggregate traffic in the system, and the time to receive the last or the k-th result tuple.

The simulator and the implementation use the same code base. The simulator allows us to scale up to 10,000 nodes, after which the simulation no longer fits in RAM – a limitation of the simulation, not of the PIER architecture itself. The simulator's scalability comes at the expense of ignoring the cross-traffic in the network and the CPU and memory utilization. We use the King model topology [32] with each node having a 1.5Mbps in-bound and out-bound link capacity. We use FIFO messaging queuing to model network traffic.

API Function	Example for AVERAGE
void $addvalue$(value)	`this.count++;` `this.sum += value;`
value $getresult$()	`return sum/count;`
void $combinePSR$(psr)	`this.count += psr.count;` `this.sum += psr.sum;`

Table 4.1: API for aggregation functions and an example implementation of an AVERAGE function.

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

For each data point, the same experiment was run ten times with the average plotted and the errorbars indicating the average plus and minus the standard deviation. To compute the network communication caused by the query workload (and discount the DHT maintenance and query dissemination traffic) a second identical experiment was performed with the query being disseminated but not executed. The network communication measurements presented are the difference between the two experiments.

4.2 Aggregation

Aggregation is the process of condensing a large collection of a data into a single value. Aggregation is commonly combined with grouping such that a set of tuples is partitioned into a number of groups and an aggregate value is reported for each group.

An aggregation function can be defined using two operations on the aggregate state: add a value to the aggregate, and get the value of aggregate. The aggregate function's running state is called a partial state record (PSR). Some aggregates such as MAX, MIN, SUM, AVERAGE are relatively straightforward, while other aggregates such as histograms and spectral Bloom filters [15] have more complex operation implementations. A special property of PSRs is that they can also be combined together using a third operation defined for the aggregation function. The API and an example are detailed in Table 4.1.

In [27] three classes of aggregate functions are defined: distributive, algebraic, and holistic. [54] expands the classification of aggregations with two additional classes: unique and content sensitive. We primarily consider distributive and algebraic aggregations which have the special property of having constant sized PSRs. Content sensitive aggregates have PSRs that are sized proportional to some statistical property of the data, such as the range or variance. Our techniques can also apply to these functions insofar as the properties of the data allow the size of the PSR to be approximated as a constant. Holistic (and unique) functions have PSRs that are proportional to the number of (unique) data items and do not benefit from our algorithms although they can still be correctly computed.

We divide our discussion of execution techniques into two groups. First we describe "one-shot" aggregates where the aggregate is computed once for a static set of tuples, and then we describe continuous queries where the aggregate is computed periodically over a dynamic stream of data. We discuss these with respect to relation R and the following generic COUNT query:

```
SELECT count(*)
FROM R;
```

4.2.1 One-Shot Aggregation Queries

The simplest method for computing an aggregate is to collect each individual source tuple in one location, and then calculate the aggregate. In this scenario, the network usage is proportional to the number of tuples, t, multiplied by the size of the aggregate field(s), s, being aggregated. The drawbacks of this method are that it requires a significant amount of communication and it concentrates the entire the *in-bandwidth* load ($t \times s$) onto a single root node. However, by taking advantage of the ability to first aggregate sets of tuples into PSRs and then aggregate those PSRs together we can develop more efficient algorithms in the spirit of systems like TAG [54].

We can easily decrease the communication by having each node aggregate its local data first, and then send that PSR to the root. The total communication bandwidth is now proportional to the number of nodes, n, in the system and the size of a PSR, p. In most cases $n \ll t$ and $p \simeq s$ so $n \times p \ll t \times s$. Therefore we only consider algorithms which communicate PSRs instead of the source tuple field.

For one-shot aggregation queries we develop eight algorithms on four design dimensions as shown in Figure 4.1. The first dimension, *structure*, determines the type of dataflow topology used: one level or multiple levels. The simplest algorithms are one level, sending all the PSRs directly to a single root node. The multi-level algorithms form a tree where each node except the root has a single parent to which it sends its PSRs to. We describe specific tree algorithms shortly.

The second dimension indicates whether there is in-network aggregation of the data. If there is no in-network aggregation each node will only forward PSRs towards the root instead of combining PSRs first and only forwarding the combined PSR. In-network aggregation only applies to multi-level topologies which have trees with interior nodes. If there is just a single level, there are only leaf nodes and one root, there are no opportunities for in-network aggregation. Multi-level topologies with no in-network aggregation require that interior tree nodes send multiple messages to their parents, since each child's PSR is forwarded individually. These algorithms are not beneficial since they involve substantially more network communication. These extra messages do not improve either of the other metrics since the latency is increased and the root will still receive a message from each node. We only consider this group of algorithms to show the benefit of in-network aggregation.

The third dimension represents the routing method. The options are to use IP routing or the DHT routing. With IP routing the query hard-codes the IP socket address where each node should forward its PSR to. This method is feasible when the query only needs to specify a single root node, which could be

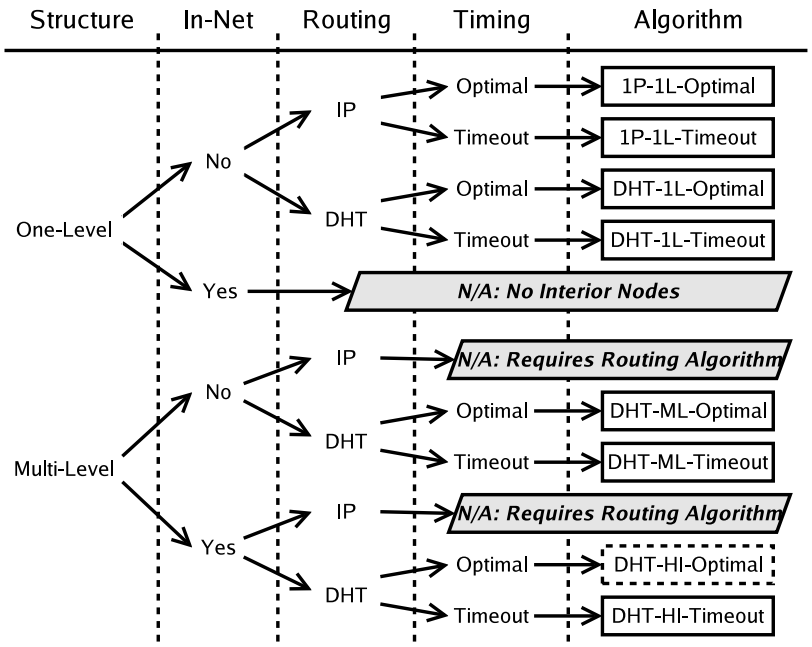


Figure 4.1: Various one-shot aggregation algorithms over four dimensions. Algorithms in dashed boxes are not evaluated via simulation because they are not implemented in PIER.

the node that issues the query. For a multi-level hierarchy, each of the interior nodes must be predetermined and assigned children. Building this topology is equivalent to using an overlay network, and requires an additional out-of-band routing algorithm. We do not propose any such algorithm since the DHT is one such routing algorithm.

When DHT routing is used the query only needs to specify a routing identifier for the root. For one-level topologies, the DHT put method is invoked, while for multi-level topologies the DHT $send$ method with upcalls is used to form a tree. The tree is dynamically formed using 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 the PSR towards the root identifier specified in the query. At the first hop along the routing path, PIER receives an upcall and processes the PSR. If combined with in-network aggregation PIER will combine the received PSR with its own local PSR. After waiting for more PSRs to arrive from other nodes, the node then forwards the PSR towards the root using the DHT $send$. If there is no in-network aggregation the PSR will be immediately forwarded towards the root using the DHT $send$. At the next hop (one step closer to the root) that PIER node repeats the same procedure. Eventually the root will receive PSRs that combine to include data from every node and the root can produce the final global answer.

Finally, the timing dimension determines how each node decides when to send its PSR to its parent (for multi-level topologies) and when the root forwards an answer to the requester (for all topologies). This is necessary because aggregation is a blocking operation and the result can not be produced until all the data has been processed. The optimal timing (which has the lowest latency without missing any PSRs) is immediately

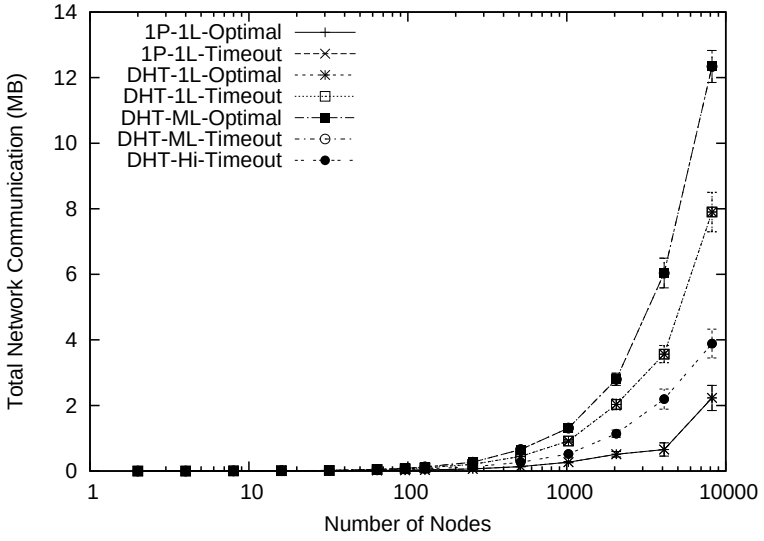


Figure 4.2: Total network communication for one-shot aggregation algorithms with varying number of nodes.

after the node receives all the PSRs from its children. For one-level topologies this is after the root receives a PSR from each node in the system. However in practice, the root will not know *a priori* how many nodes are in the system and so the optimal solution is often unattainable. Instead, we can use a query-specified timeout condition to determine when to send. A simple condition is a fixed timeout, such as five seconds. The condition can also be specified as a rate, such as when the node receives less than one PSR per five seconds. The timeout strategy attempts to guess at when all the data has been received. If the timeout occurs too soon, the node will be required to send an additional PSR with the additional data. If the timeout occurs late, the end-user latency is unnecessarily increased.

Using these four dimensions we can create eight algorithms, seven of which can be directly implemented in PIER using different query plans. DHT-Hi-Optimal is not implemented in PIER since the topology generated by the DHT is not known *a priori*. Without knowing the topology the expected number of messages to receive at each node can not be predetermined. A brief description of each algorithm can be found in Appendix B.

Figure 4.2 shows that overall bandwidth usage for the seven implemented algorithms for various sized networks and a timeout condition of five seconds. The IP-1L-Optimal always has the least overall bandwidth usage. This is expected as each node sends exactly one PSR except the root which sends a single result tuple. The IP-1L-Timeout also has low overall bandwidth usage, however in larger networks the timeout is too short, so some PSRs arrive after the timeout. The root must then revise the answer multiple times resulting in slightly higher communication costs (the difference is too small to be noticeable in the figure). The DHT-1L-Optimal and DHT-1L-Timeout algorithms require more bandwidth that IP-1L-Optimal and IP-1L-Timeout due to the DHT message overheads.

The DHT-Hi-Timeout falls in the middle of graph since the interior nodes will send at least two

PSRs. The cause of this extra communication is due to the lack of topology information (nodes do not know their depth in the tree) and a single timeout condition is imposed on all nodes. In the DHT-Hi-Optimal, nodes would know the topology and would be able to determine if they were leaf nodes. Leaf nodes do not require a timeout since they do not receive PSRs from other nodes. Therefore leaf nodes should send their PSR immediately. Interior nodes with only leaf nodes as children would then send their PSRs next (or have the next lowest timeout) and so forth up the tree to the root which would send last (or have the highest timeout).

Without this information, every node must first act as if it were a leaf node and send its local PSR towards the root. Nodes that receive data from their children now know they are interior nodes and will send a second PSR based on the data received from their children. Interior nodes higher in the tree will receive the second round of PSRs and send a third round of PSRs. Eventually only the root will receive additional PSRs and the final answer will be produced. The total number of message sent is based on the number of nodes at each level of the tree.

The DHT-ML-Optimal and DHT-ML-Timeout algorithms illustrate the cost of not using in-network processing. Both algorithms use significantly more bandwidth since each node's PSR is simply forwarded to root and not combined with other PSRs.

In Figure 4.3 we can see that in-network processing (DHT-Hi-Timeout) has the distinct advantage of significantly reducing the bytes received at the root and distributing that load to other nodes in the system. The nodes receiving additional load may be at any level (except the leaves) in the tree, since the load is based on the number of direct children, not the size of the subtree. Multi-level algorithms with no in-network processing also cause other nodes to receive additional messages but do not change the load on the root. Nodes that are higher in the tree (culminating with the root) receive more messages than nodes lower in the tree. Single level algorithms simply have high load at the single root.

The final metric we consider for these algorithms is latency for the final answer to be received, as shown in Figure 4.4. This metric allows us to see the big difference between the optimal timing and timeout methods. Algorithms that have optimal timing produce final answers earlier than those that must wait for a timeout to occur. This is particularly noticeable with small networks and less noticeable with larger networks. Furthermore, in very large networks the timeout may be too short, which causes the initial results to be revised later. While these early results do have some value they also use more network communication. This highlights the general problem with timeouts which are fixed and are a "one size fits all" solution. For continuous queries that we discuss next, we are able to improve timeouts by adapting to the conditions of the network and topology.

In summary, hierarchical (multi-level) in-network processing aggregation query plans require more overall bandwidth but are better at spreading that load more fairly. The timing method not only effects overall bandwidth, since early timeouts cause extra messages, but also latency is dictated by timing. While optimal timing is (not surprisingly) best, it is often not feasible since the number of participating nodes and/or topology are not known *a priori*.

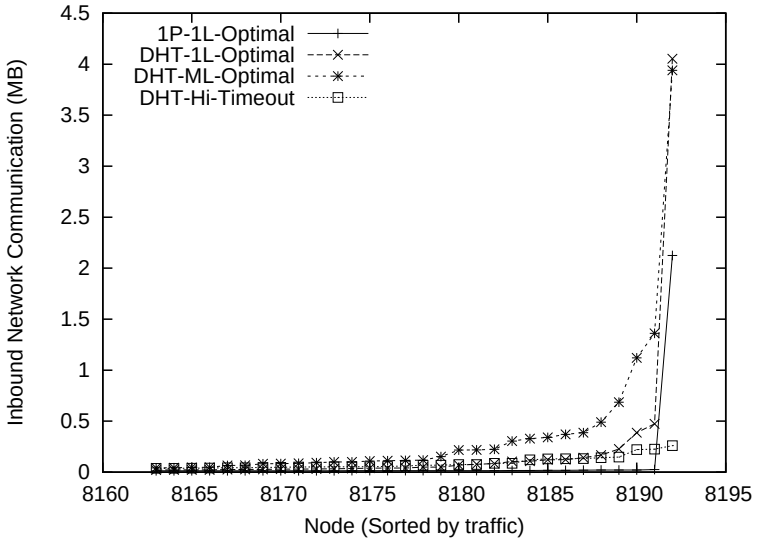


Figure 4.3: In-bound bandwidth usage for the fifty nodes receiving the most messages in a single one-shot aggregation experiment with 8192 nodes.

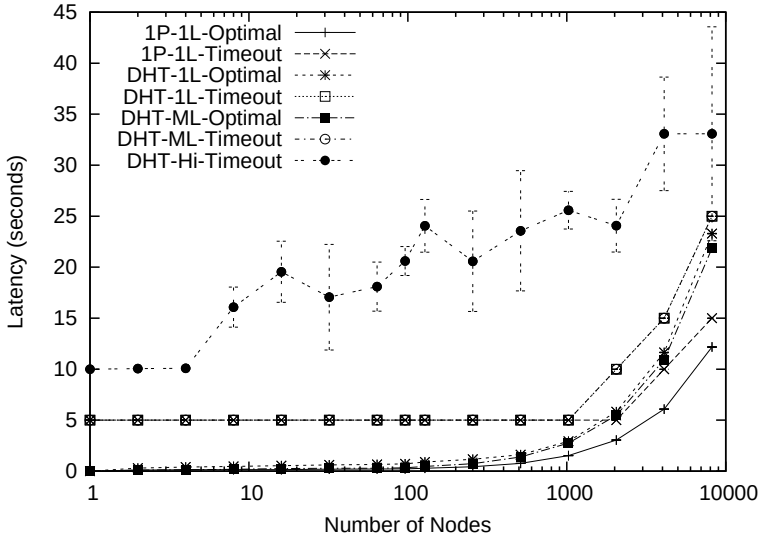


Figure 4.4: Latency for one-shot aggregation queries with varying number of nodes.

4.2.2 Continuous Aggregation Queries

While the semantics of continuous queries are an entire research topic (see [12, 3]), PIER adopts a simple model for continuous aggregation queries. Each query specifies the frequency of epochs with the first epoch coinciding with the start of the query. For each epoch, the query results are solely based on the data fetched from the data source at the start of the epoch. It is the responsibility of the data source to only provide current/valid data. This is closest to semantics commonly referred to as *non-overlapping sliding windows*. Other query semantics are possible using the built-in query operators. For example, *overlapping sliding windows* can be achieved through the use of the MemUpdateCache operator (see Section 3.3.5) which stores tuples for a specified duration of time.

With respect to the distributed aggregation computation, continuous queries can be seen as issuing a one-shot query periodically with an extra field containing an incrementing epoch number. A straw-man implementation of executing a continuous query is to simply execute every epoch as a separate query. However, the primary difference to one-shot queries is that the query engine knows that the query will be essentially "issued" periodically. Thus the system can store the topology and timing from the previous epoch to use again or adapt in the next epoch. This enables a wider range of possible algorithms/query plans. Some of the new query plans can provide lower latency and/or decreased communication as compared to the straw-man implementation.

With one-shot queries we identified four dimensions for aggregation algorithms (structure, in-network computation, routing, and timing). For continuous queries we add a new dimension, dynamism. Dynamism indicates whether the topology changes constantly between epochs or if the topology is held fixed (or stable) after the first epoch except to recover from failures. Normally, the DHT is prone to changing routes over time, sometimes changing with each message. This is because the DHT router is trying to greedily optimize end-to-end latency[1]. When a message is sent on one path, that path may become more expensive by the time the next message is sent and another path may now be faster. For many applications this is the desired behavior, however, for applications that want to use the same path for multiple messages, the application layer must store the path and force the DHT to use that path. This is not the same as source routing, since each node only stores and remembers the next hop. Instead it is more similar to network protocols that use virtual channels.

Continuous queries also enable us to solve a problem that one-shot queries suffer. The DHT routing may create trees that are not well balanced or that have very high in-degree for a few nodes. In one-shot queries we noticed that if one node, such as the root, has a very high in-degree, the latency is longer. The DHT may create trees that have nodes (maybe not the root) that also have high in-degree. Recall Figure 4.3, which shows the in-bound network communication (which is proportional to the in-degree since all PSR messages are of equal size) for some nodes during a one-shot query. The DHT-based hierarchical aggregation (DHT-Hi-Timeout) query shows that three nodes receive over five times the data of most nodes and another six nodes receive over twice the data of the remaining 8180 nodes. This imbalance of in-degree and network

[1] This is not a requirement of a DHT implementation, rather this is often a design choice made by DHT designers.

traffic can lead to network congestion and increased latency.

With a one-shot query there is no opportunity to correct the imbalance. However, with continuous queries an in-degree imbalance in one epoch can be corrected in future epochs, improving the performance of the query over time. Nodes with high in-degree can send a message to some of their children instructing them to use a different parent. Over a number of epochs the tree will stabilize to a new topology that enforces a maximum in-degree policy. While the forming of this tree requires additional messaging, continuous queries enable the amortizing of this cost over the length of the query. We call this method *tree* routing.

In PIER we use a simple implementation to enforce the maximum in-degree. When a node sends a message to a parent, it appends two fields to the message; one field that includes the node's height in the tree and a second field advertising the number of additional children that the node and all of its children can handle. For a leaf node, the number of children that the node can handle is a fixed number set to 64 by default. For an interior node, that number is the sum of the values most recently reported by its children plus its fixed value minus the number of active children. The height of a node is calculated by adding 1 to the maximum height value reported by any active child, or 0 if the node does not have any active children.

When a node receives a message from a child it determines whether this child is in its list of active children. If the node is in the list, the record for the child is updated to reflect the information in the latest message received. If the child is not in the list and there is a free slot at this node, the child is simply added to the list active children. However, if the child is new and there are no free slots at this node, the node will select a replacement parent for the child. The replacement parent is chosen by selecting the active child with the lowest height and the highest number of advertised free slots. Once the replacement is chosen the parent sends a redirect message to the child. Upon receiving the redirect message, the child will now send messages to the new parent until it stops responding or sends the child another redirect message.

Finally, we also have two additional timing strategies for continuous queries, *learned* and *topology*. In the learned scheme, interior tree nodes maintain a list of their children across epochs. Once a node receives data for the current epoch from each of those nodes it can then send its PSR to its parent. If the tree is stable, after the first epoch the timing will be optimal. In practice the tree may not be stable, a node that does not receive data from a child after a fixed timeout (i.e. the length of an epoch) the child is removed from the list and the received data is sent. If a node receives a PSR from a new node, it is immediately added to the children list. This method could be further improved by replacing the fixed timeout with an adaptive timeout based on the each child's average latency and the variance. The fixed timeout is used in our experiments.

The topology timing method is very similar to the learned method. The one difference is when a node instructs a child to use a new parent, it immediately removes that child from its child list instead of waiting for the fixed timeout. This is strictly better than the learned method which will incur unnecessary timeouts, increasing latency while the tree is being optimized. This method is only applicable if the tree routing method is used.

Using these five dimensions we can create a plethora of algorithms many of which are implemented in PIER. Figures 4.5, 4.6 and 4.7 show the options. In Appendix C each of the algorithms is briefly described. We now show a number of experiments to highlight the important differences. In all experiments we continue

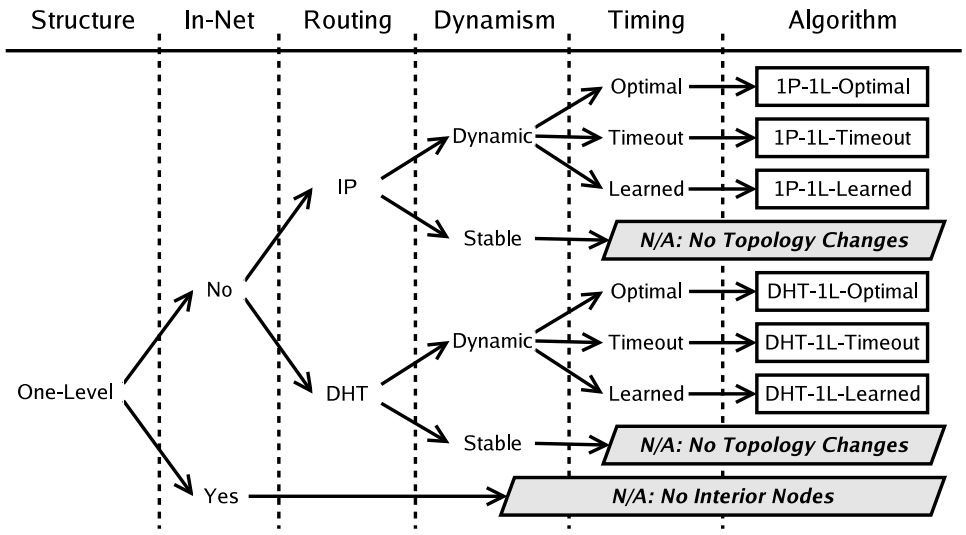


Figure 4.5: Various continuous aggregation query algorithms over five dimensions.

to use a simple COUNT aggregation query with a new epoch every five seconds and a total of 30 epochs for a total of five minutes. All timeouts are also five seconds. We examine the total network communication and latency for a system with varying number of nodes.

In the first set of simulations we explore the difference in the timing dimension when all data is sent directly to the root. The root is the only node making a timing decision, which is when to produce the result. In Figure 4.8 we can see that in this scenario, timing has no discernible impact on network communication, and the only difference is between the algorithms that use direct IP communication and the DHT routing. The direct IP methods show near linear growth in the network communication, while the DHT methods show faster than linear growth because of the DHT routing overheads. Figure 4.9 shows the clear differences between the three timing methods. Regardless of the routing, optimal and learned perform the same, with timeouts incurring additional latency.

The learned timing method is effective in determining the root's children and triggering result generation once all the data has been received. This is good since the optimal timing method is not practical in real systems where the complete topology for all epochs is not known *a priori*. We no longer show results for the timeout method since it is strictly worse than learned method.

We next examine the differences between the structure of communication and the use of in-network aggregation. As with the one-shot queries Figure 4.10 shows that sending the PSRs directly to the root using IP is optimal with respect to overall network communication. DHT-ML-Learned used the most bandwidth since every node sends its PSR to each node on the route to the root, however at those nodes the PSRs are not combined. In this situation the same PSR is being sent multiple times resulting in excessive network traffic. The hierarchical aggregation, DHT-Hi-Learned, while not as good as optimal, performs well. Surprisingly, sending the PSRs directly to the root also using a large amount of bandwidth even though each

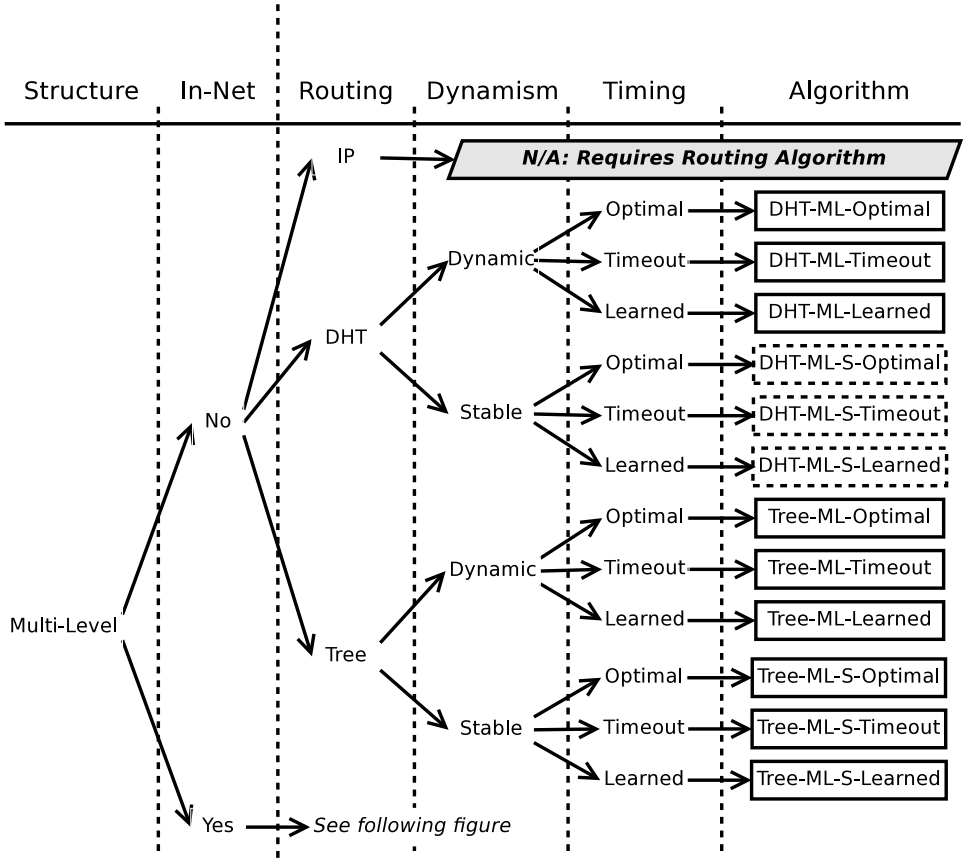


Figure 4.6: Various continuous aggregation query algorithms over five dimensions. Algorithms in dashed boxes are not evaluated via simulation because they are not implemented in PIER.

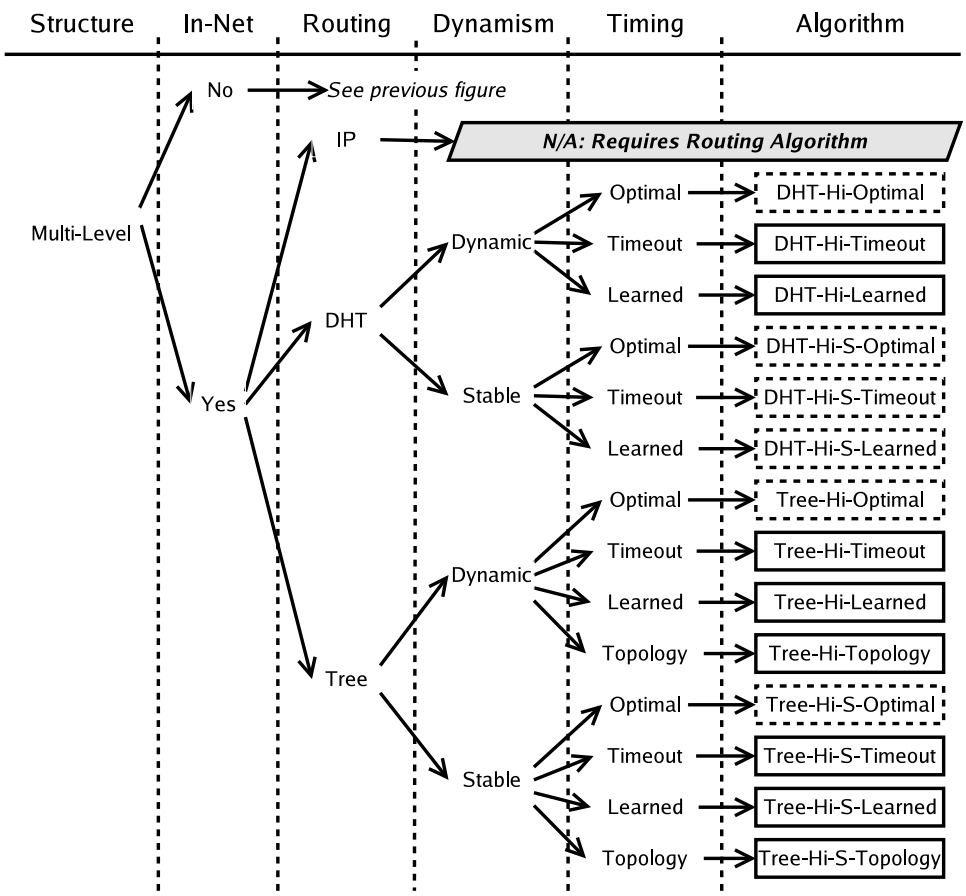


Figure 4.7: Various continuous aggregation query algorithms over five dimensions. Algorithms in dashed boxes are not evaluated via simulation because they are not implemented in PIER .

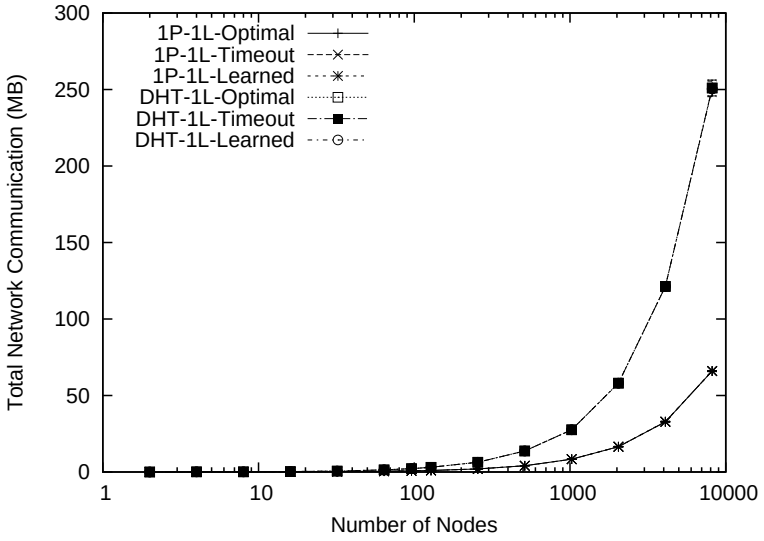


Figure 4.8: Total network communication for the routing/timing dimensions of continuous aggregation algorithms with varying number of nodes.

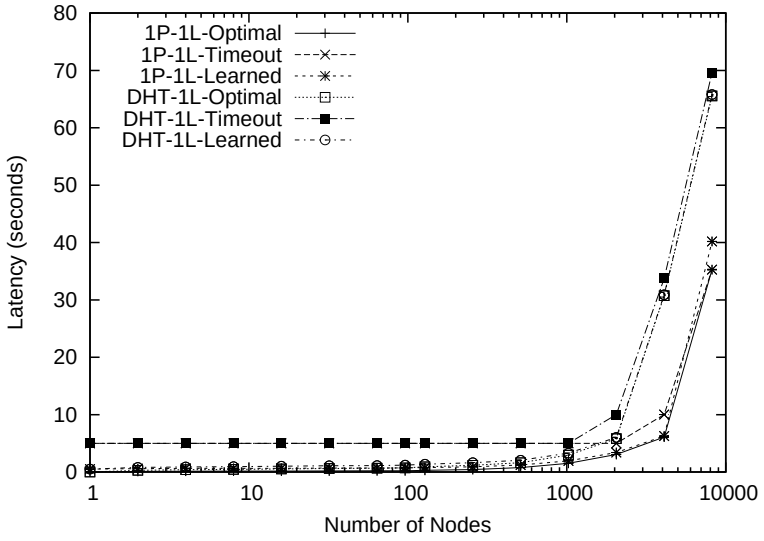


Figure 4.9: Average latency for the routing/timing dimensions of continuous aggregation algorithms with varying number of nodes.

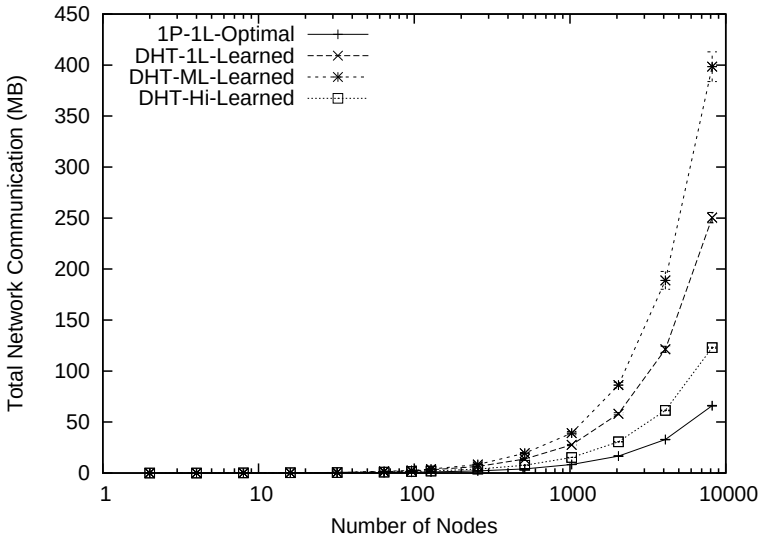


Figure 4.10: Total network communication for the structure/in-network dimensions of continuous aggregation algorithms with varying number of nodes.

PSR is only sent once. The hierarchical method outperforms sending the PSRs directly to the root because so many messages are routed to the root that the root's in-bound network becomes congested. When a node is congested the DHT may incur timeouts and believes messages were lost, and will retry sending the message using additional bandwidth.

With respect to latency, the hierarchical method outperforms all other methods when the network size is very large, including the IP-based method with optimal timing as shown in Figure 4.11. This can be easily explained by considering the amount of the data the root and other other nodes are receiving. In Figure 4.12 we see that the distribution of network load is highly skewed as before. With the exception of hierarchical aggregation the root can become heavily congested which causes the high latency.

While hierarchical aggregation using the DHT was the best at distributing load, as previously shown in Figure 4.12 there are still over a dozen nodes receiving the bulk of the network traffic. In the last set of experiments we explore the value of further optimizing the DHT's aggregation tree by limiting a node's in-degree.

In Figure 4.13 we see that IP-1L-Optimal still performs the best with respect to the total network communication. The Tree-Hi-Topology actually uses more bandwidth than the DHT-Hi-Learned because of the extra messages sent from parents to the children. However, we can reduce the number of these extra messages using the Tree-Hi-S-Topology method which prevents the tree from changing between epochs unless a failure is detected. This method, even with some extra messages, uses less network communication than the standard hierarchical method (DHT-Hi-Learned) because there are no longer any nodes with in-bound congestion and therefore fewer retries for messages. Figure 4.14 clearly shows that the tree routing strategy results in an even distribution of the in-bandwidth to all nodes.

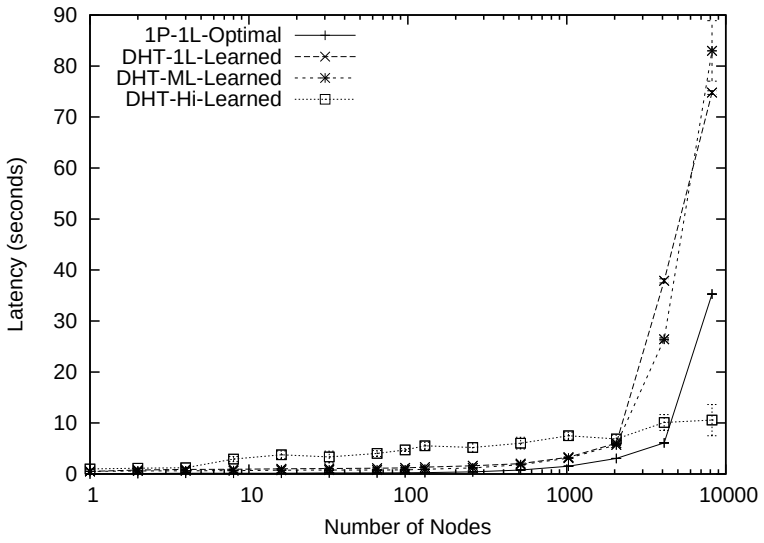


Figure 4.11: Latency for the structure/in-network dimensions of continuous aggregation algorithms with varying number of nodes.

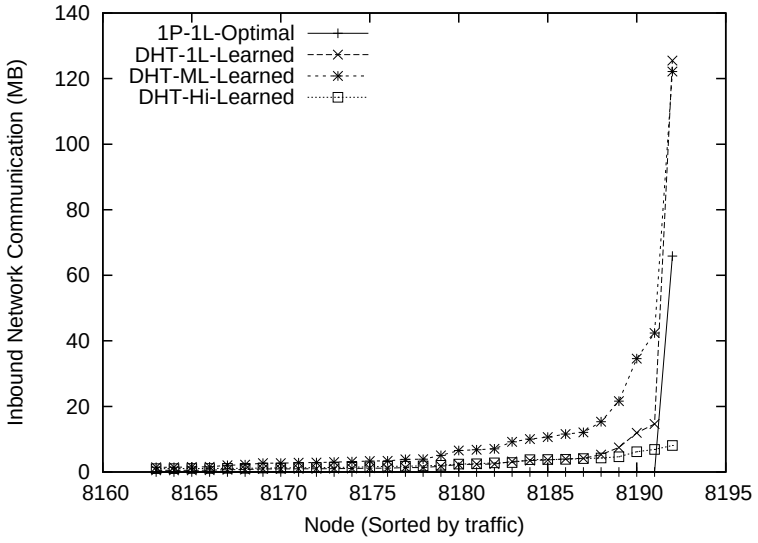


Figure 4.12: In-bound network communication for the structure/in-network dimensions of continuous aggregation algorithms for a single experiment with 8192 nodes.

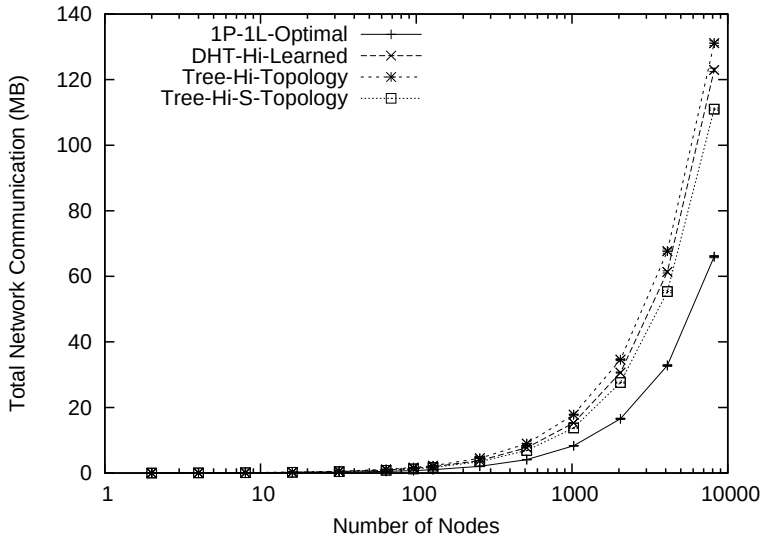


Figure 4.13: Total network communication for the dynamics dimension of continuous aggregation algorithms with varying number of nodes.

While Tree-Hi-S-Topology is unable to match the optimal communication, the longer the query runs with a stable set of participating nodes the smaller the gap between the two strategies will be. This is because most of the extra communication is occurring in the first few epochs while the tree is being optimized. By the fourth epoch the tree is stable and with the exception of the DHT message overheads is optimal.

Figure 4.15 reinforces the desire to reduce in-bound congestion. The latency for Tree-Hi-S-Topology is the lowest when the network size is largest. The other hierarchical strategies have slightly higher latencies. The latency curves for three DHT and tree based strategies are not monotonically increasing as the number of nodes in the system increases because of the randomness in the tree structure based on the DHT routing. Latency is based on the slowest route used from any leaf node to the root node.

Overall the Tree-Hi-S-Topology strategy offers the best set of attributes. The combination of five design choices, multi-level, in-network aggregation, DHT routing with in-degree optimization, learned timing and preventing topology changes except in failures all work together to produce a solution with low-latency, moderate overall network communication, and even distribution of network load.

4.3 Joins

Our join algorithms are adaptations of textbook parallel and distributed schemes, which leverage DHTs whenever possible. This is done both for the software elegance afforded by reuse, and because DHTs provide the underlying Internet-level scalability and robustness we desire. We use DHTs in both of the senses used in the literature – as “content-addressable networks” for routing tuples by value, and as hash tables for storing tuples. In database terms, DHTs can serve as “exchange” mechanisms [26], as hash indexes, and

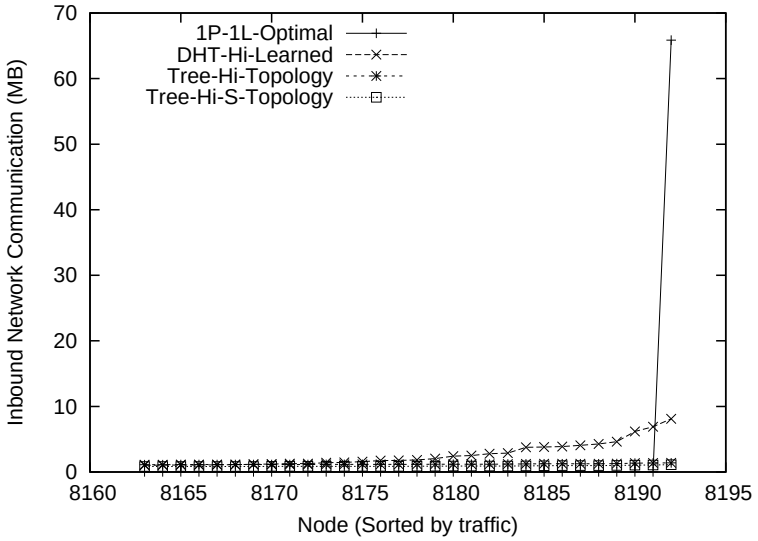


Figure 4.14: In-bound network communication for the dynamics dimension of continuous aggregation algorithms for a single experiment with 8192 nodes.

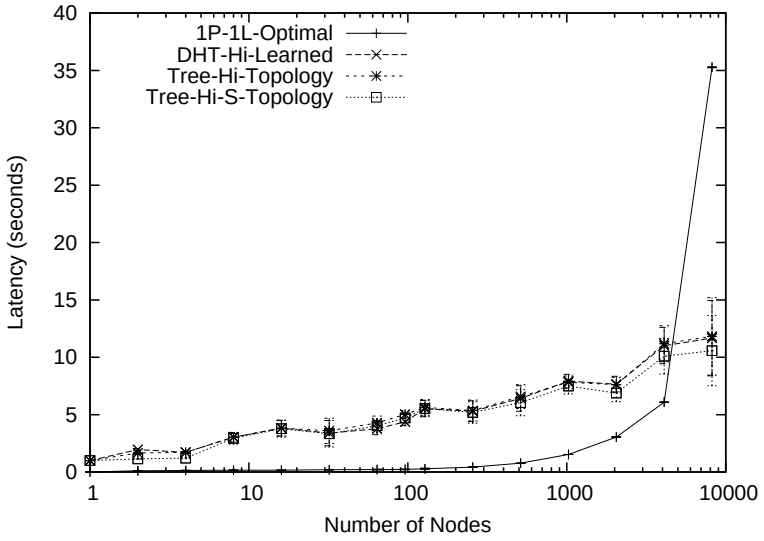


Figure 4.15: Latency for the dynamics dimension of continuous aggregation algorithms with varying number of nodes.

as the hash tables that underlie many parallel join algorithms. DHTs provide these features in the face of a volatile set of participating nodes, a critical feature not available in earlier database work. We also use DHTs to route messages other than tuples, including Bloom filters.

We have implemented two different binary equi-join algorithms, and two bandwidth-reducing rewrite schemes. We discuss these with respect to relations R and S and the following generic query:

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

We assume that the tuples for R and S are horizontally partitioned across the network. Unless specifically stated, we do not make any assumptions as to how the source data is placed throughout the network or which access handle is used to retrieve the source data.

4.3.1 Core Join Algorithms

Our most general-purpose equi-join algorithm is a DHT-based version of the pipelining *symmetric hash join* [82], interleaving building and probing of hash tables on each input relation. To begin the join PIER will first retrieve each relation, R and S, at each node from the local data source. Each tuple that satisfies all the local selection predicates is copied (with only the relevant columns remaining) and is added to a new unique DHT namespace, N_Q, using the DHT put command. The values for the join attributes are concatenated to form the resourceID for the copy. The tuples are also tagged with their source table name since their DHT name does not include the table name (the same namespace is used for all tuples from both input relations).

By the end of the join, tuples from both R and S will have been hashed on the join attribute. Tuples from both input relations (R and S) that have the same values for the join attributes will be assigned the same location identifier. This ensures that the tuples will be routed to the same node during the put operation.

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

The second join algorithm, *Fetch Matches*, is a variant of a traditional distributed join algorithm that works when one of the tables, say S, is already hashed on the join attributes. In this case, table R is locally retrieved, and for each R tuple a get is issued for the corresponding S tuple. Note that local selections on S do not improve performance – they do not avoid gets for each tuple of R, and since these gets are done

at the DHT layer, PIER's query processor does not have the opportunity to filter the S tuples at the remote site (recall Figure 3.1). In short, selections on non-DHT attributes cannot be pushed into the DHT. This is a potential avenue for future streamlining, but such improvements would come at the expense of "dirtying" DHT APIs with PIER-specific features – a design approach we avoided in our implementation. Once the S tuples arrive at the corresponding R tuple's site, predicates are applied, the concatenation is performed, and results are passed along as above.

4.3.2 Join Rewriting

Symmetric hash join requires hashing both tables, and hence can consume a great deal of bandwidth. To alleviate this when possible, we also implemented DHT-based versions of two traditional distributed query rewrite strategies, to lower the bandwidth of the symmetric hash join by avoid communicating tuples that will not join with any tuples from the other relation. Our first is a *symmetric semi-join*. In this scheme, we minimize initial communication by locally projecting both R and S to their location identifier and join keys, and performing a symmetric hash join on the two projections. The resulting tuples are pipelined into Fetch Matches joins on each of the tables' location identifiers. (In our implementation, we actually issue the two joins' fetches in parallel since we know both fetches will succeed.) Essentially this method creates two indices on the fly, one for each relation.

The other rewrite strategy uses Bloom joins. First, Bloom filters are created by each node for each of its local R and S fragments, and are published into a small temporary DHT namespace for each table. At the sites in the Bloom namespaces, the filters are OR-ed together and then broadcast to all nodes storing the opposite table. Following the receipt of a Bloom filter, a node begins retrieving its corresponding table fragment, but rehashing only those tuples that match the Bloom filter. Since Bloom filters only generate false positives (a tuple matches the filter but will not find any tuples to join with) and no false negatives, any errors introduced by using the Bloom filters results in less bandwidth savings, but does not effect correctness.

4.3.3 Evaluation of Join Strategies

For these experiments we follow the same simulation setup as described in Section 4.1. Tables R and S are synthetically generated. Unless otherwise specified, each tuple in R is padded to be 1024 bytes, each tuple in S is 1536 bytes and each result tuple is 2048 bytes. The constants in the predicates ($R.num1 > constant1$ and $S.num1 > constant2$) are chosen to produce a selectivity of 50%. In addition, the last predicate uses a function $f(R.num2, S.num2)$; since it references both R and S, any query plan must evaluate it after the equi-join. We choose a function as opposed to simply directly comparing the attribute from each of the two relations (i.e. $R.num2 > S.num2$) because it allows us to generate one set of relations for multiple tests and vary the selectivity by setting the constant in the query. We choose the distribution of the join columns such that 90% of R tuples have two matching join tuples in S (before any predicates are evaluated) and the remaining 10% have no matching tuples in S. These values where arbitrarily chosen so that most tuples are used in computing the join result. However, by having some tuples not used in the join,

the experiments will expose some differences in how the join strategies decide what data to move around in the network. For each node in the network 100 R and 100 S tuples are injected into the system prior to the query running.

As with aggregation we focus on two key metrics: network communication and latency. However, unlike aggregation, we measure latency when the 100th result tuple is received instead of the last tuple. The value 100 was chosen to be a little after the first tuple received, and well before the last. We avoid using the first response as a metric here, on the chance that it is generated locally and does not reflect network limitations. We are not interested in the time to receive the last result, because as we increase the network size and data set, we increase the number of results; at some point in that exercise we end up simply measuring the (constant) network capacity at the query site, where all results must arrive.

Centralized vs. Distributed Joins

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

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

When there is only one computation node in a 2048-node network, one would need to provision for a very high link capacity in order to obtain good response times. For instance, even if we are willing to wait one minute for the results, one needs to reserve at least 137Mbps for the downlink bandwidth, which would be very expensive in practice.

IP vs. DHTs

The traditional implementations of distributed symmetric hash joins utilize an Exchange-like [26] operator. These operators hash and route tuples with knowledge and IP addresses of every participating node. While this is not practical in Internet-scale systems it is interesting to compare the performance of a DHT-based solution to an idealized IP-based solution.

In Figure 4.16 we show the network overhead of the DHT-based solution vs. an IP-based routing solution. In Figure 4.17 we plot the latency of the two options. Not surprisingly the IP method uses less network bandwidth and is faster. When the size of the network exceeds eight nodes the latency till the 100th result tuple begins to decrease for both methods. This is caused by the increase in the number of result tuples (recall that the number of source tuples in the system is proportional to the number of the nodes in the system) and therefore the 100th result is produced sooner. The additional bandwidth used by the lookup messages in the DHT's put method accounts for the addition communication and latency as each tuple is only sent

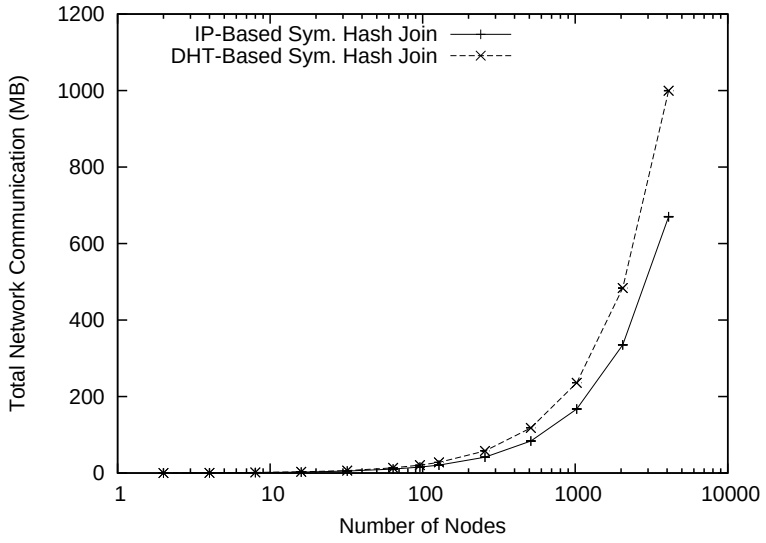


Figure 4.16: Overall bandwidth usage for a symmetric hash join with varying number of nodes.

across the network once in both scenarios. However, while the DHT has an impact on latency and network communication, using the DHT allows us to issue a query without *a priori* knowledge of every node in the system and their IP addresses.

Join Strategies

We now evaluate the four join strategies when the selectivity of the predicate on the S relation is varied. In Figure 4.18 the overall in-bound network communication is shown. As expected, the symmetric hash join uses the most network resources for many workloads since both tables are rehashed. The increase in the total in-bound traffic is due to the fact that both the number of tuples of S that are rehashed and the number of results increase with the selectivity of the selection on S. In contrast, the Fetch Matches strategy basically uses a constant amount of network resources because the selection on S cannot be pushed down in the query plan (the increase shown is due to sending the results generated by the query). This means that regardless of how selective the predicate is, the S tuples must still be retrieved and then evaluated against the predicate at the computation node. In the symmetric semi-join rewrite, the second join transfers only those tuples of S and R that match. As a result, the total in-bound traffic increases linearly with the selectivity of the predicate on S. Finally, in the Bloom filter case, as long as the selection on S has low selectivity, the Bloom filters are able to significantly reduce the rehashing on R, as many R tuples will not have an S tuple to join with. However, as the selectivity of the selection on S increase, the Bloom filters are no longer effective in eliminating the rehashing of R tuples, and the the algorithm starts to perform similar to the symmetric hash join algorithm.

In Figure 4.19 the latency till the 100th tuple is plotted. The latency is mostly constant across all the workloads shown. The latency is determined by the stages of the join. Each strategy requires distributing

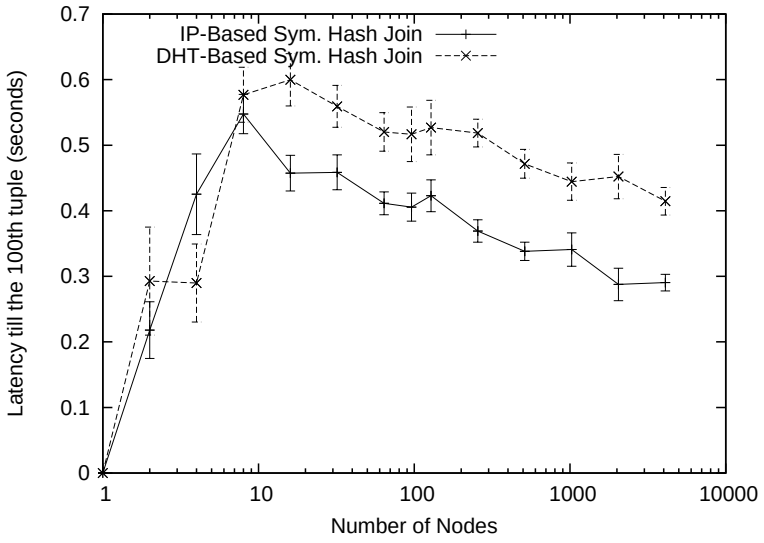


Figure 4.17: Latency till the 100th result for a symmetric hash join with varying number of nodes.

the query to the participating nodes and the delivery of the results (direct IP communication between nodes). In the symmetric hash join, the DHT must route a lookup message, wait for a lookup response to determine the destination of each tuple, and then send the tuple directly to that node. Fetch Matches is slightly faster since the response to the lookup message includes the tuple and does not require another message (recall Section 3.2.4. The symmetric semi-join rewrite is essentially a symmetric hash join followed by a Fetch Matches join so its latency is almost the sum of the two strategies minus the common overhead of query dissemination and result collection. Finally the Bloom filter rewrite has the highest latency since the creation and dissemination of the Bloom filters must occur before the tuples are rehashed.

4.3.4 Hierarchical Joins

Like hierarchical aggregation, the goal of hierarchical joins is to reduce the communication load. In this case, we attempt to reduce the *out-bandwidth* of some nodes rather than the in-bandwidth as was the case with hierarchical aggregation. When performing a join the size of the output relation is not necessarily negligible. In fact it is possible that the size of the output relation is significantly larger than the input relations. The output of a join can be the cross product of the input relations, or have a maximum of $|R| \times |S|$ tuples. In this case, nodes have the burden of both receiving input tuples and sending output tuples over the network. In this section we attempt to distribute the output load among as many participating nodes.

In the partitioning ("rehash") portion of a parallel hash join, source tuples can be routed through the network (using the DHT $send$), destined for the correct hash bucket on some node. As each tuple is forwarded along the path, each intermediate node intercepts it using the DHT upcall callbacks, caches a copy, and annotates it with its local node identifier before forwarding it along. When two tuples cached at the

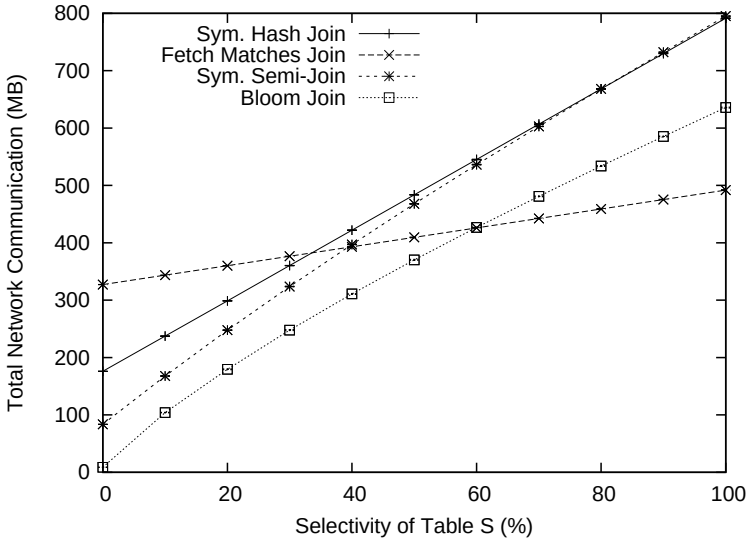


Figure 4.18: Overall bandwidth usage for four join strategies with varying selectivity on table S.

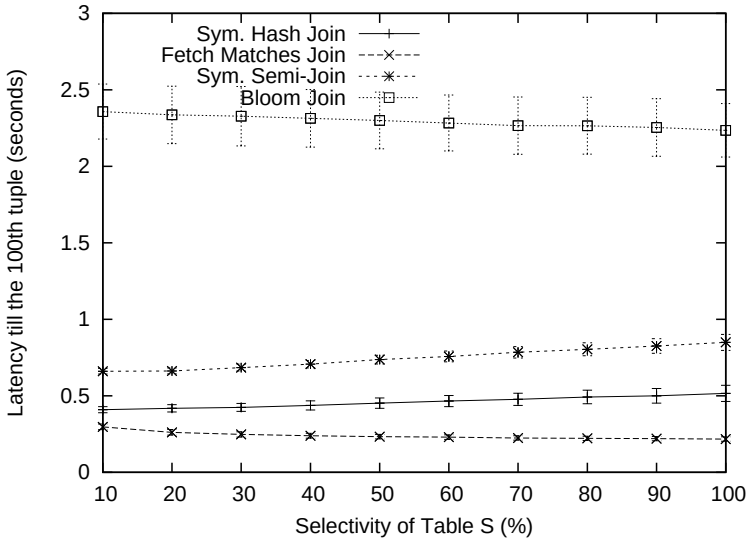


Figure 4.19: Latency till the 100th result for four join strategies with varying selectivity on table S.

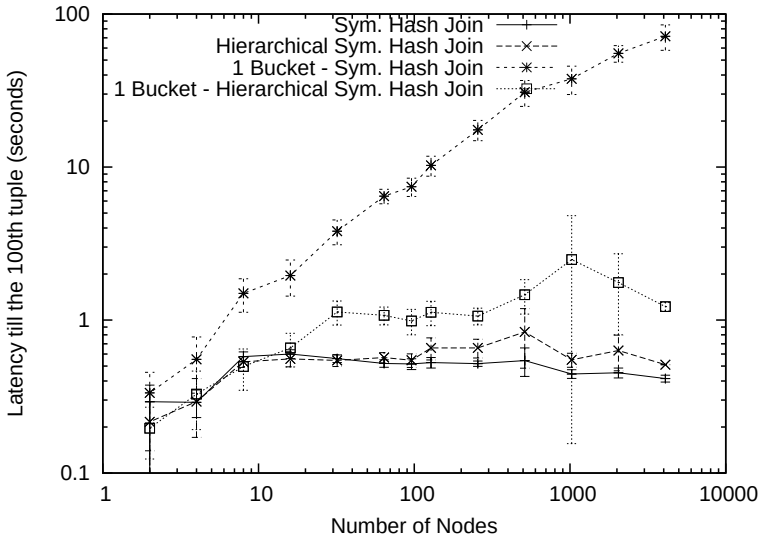


Figure 4.20: Latency till the 100th result for a symmetric hash join with varying number of nodes.

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", before matching tuples have reached the node responsible for their respective hash-bucket. This could potentially improve latency and shift the out-bandwidth load from the node responsible for a hash-bucket to nodes along the paths to that node.

With respect to out-bandwidth load, the worst case scenario is caused by a skewed workload that only has one hash bucket which is therefore assigned to a single node. The network bottleneck at that node could be 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. Unfortunately our results show this is not feasible.

In our experiments we show four different methods, the basic symmetric hash join, the hierarchical symmetric hash join, and both methods when all tuples are assigned to the same hash-bucket to simulate the worst-case workload. Figure 4.20 shows the the hierarchical methods reduce the latency to under one second. However, as shown in Figure 4.21, the decrease in latency comes at a significant increase in overall network communication.

Hierarchical joins require sending the entire tuple to each node on the path to the hash bucket. In the standard join the DHT only routes a small message along the path to discover the destination node and then sends the payload directly to the node. The difference in the size of the lookup message (about 124 bytes) and the entire tuple (between 1100 and 1620 bytes) cause the significant increase in network traffic.

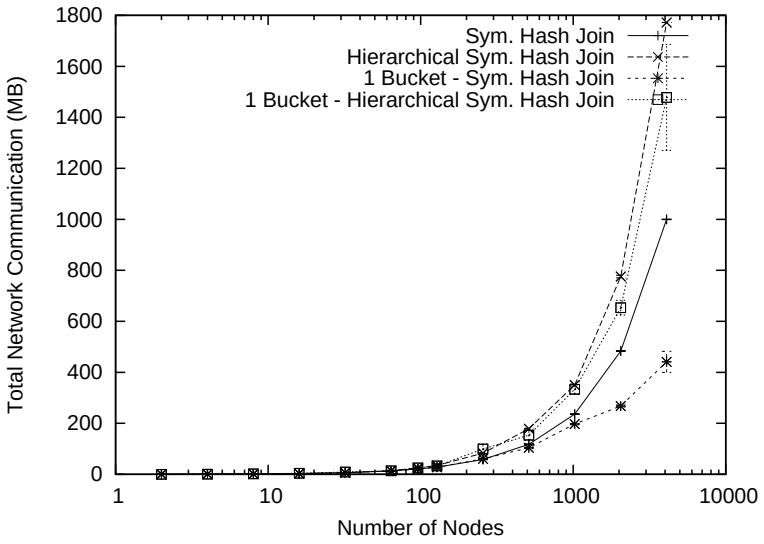


Figure 4.21: Overall bandwidth usage for a symmetric hash join with varying number of nodes.

Chapter 5

Multi-Query Optimization

In the previous chapter we described various methods for executing a single aggregation query. We showed that learning and optimizing the communication tree over time improves latency, decreases overall network communication, and more equally distributes in-bound network communication. However, an important challenge in modern distributed querying is to efficiently process multiple continuous aggregation queries simultaneously. Processing each query independently may be infeasible due to network bandwidth constraints. We now focus on further decreasing overall network communication through multi-query optimizations which will utilize the single aggregation query execution methods previously discussed.

In this chapter, we consider this problem in the context of distributed aggregation queries that vary in their selection predicates. We identify scenarios in which a large set of q such queries can be answered by executing $k \ll q$ *different* queries. The k queries are revealed by analyzing a boolean matrix capturing the connection between data and the queries that they satisfy, in a manner akin to familiar techniques like Gaussian elimination. Indeed, we identify a class of *linear* aggregate functions (including SUM, COUNT and AVERAGE), and show that the sharing potential for such queries can be optimally recovered using standard matrix decompositions from computational linear algebra. For some other typical aggregation functions (including MIN and MAX) we find that optimal sharing maps to the NP-hard *set basis* problem. However, for those scenarios, we present a family of heuristic algorithms and demonstrate that they perform well for moderately-sized matrices. We also present a dynamic distributed system architecture to exploit sharing opportunities, and experimentally evaluate the benefits of our techniques via a novel, flexible random workload generator we develop for this setting.

5.1 Overview

The goal of the algorithms we present in this chapter is to minimize overall network communication. During an aggregation query, each node must send a partial state record (PSR) to its parent in an aggregation tree. If there is no sharing, then we are communicating one partial state record (PSR) per node

per query per epoch. If we have q queries, our goal is to only send k PSRs per node per epoch, where $k \ll q$, such that the k PSRs are sufficient to compute the answer to all q queries. The next section discusses the intuition for how to select these k PSRs.

5.1.1 The Intuition

Consider a very simple example distributed monitoring example system with n nodes. Each of the nodes examines its local stream of packets. Each packet is annotated with three boolean values: (1) whether there is a reverse DNS entry for the source, (2) if the source is on a spam blacklist, and (3) if the packet is marked suspicious by an intrusion detection system (IDS). One could imagine various applications monitoring all n streams at once by issuing a continuous query to count the number of global "bad" packets, where each person determines "bad" as some predicate over the three flags. Here are example query predicates from five COUNT queries over the stream of packets from all the nodes:

1. `WHERE noDNS = TRUE`
2. `WHERE suspicious = TRUE`
3. `WHERE noDNS = TRUE OR suspicious = TRUE`
4. `WHERE onSpamBlackList = TRUE`
5. `WHERE onSpamBlackList = TRUE AND suspicious = TRUE`

We use an idea from Krishnamurthy, et al. [44] to get an insight for how to execute these queries using fewer than 5 PSRs. In their work, they look at the set of queries that each tuple in the stream satisfies, and use this classification to partition the tuple-space to minimize the number of aggregation operations (thereby reducing computation time). Returning to our five example queries above, suppose in a single epoch at node i we have tuples that can be partitioned into exactly one of the following five categories:

1. Tuples that satisfy queries 1 and 3 only
2. Tuples that satisfy queries 2 and 3 only
3. Tuples that satisfy query 4 only
4. Tuples that satisfy queries 1, 3, and 4 only
5. Tuples that satisfy queries 2, 3, 4 and 5 only

We will refer to each of these categories as a *fragment*. As a compact notation, we can represent this as a ($f \times q$) boolean *fragment matrix*, F, with each column representing a query (numbered from left to

right) and each row representing a fragment:

$$F = \begin{array}{c} \text{Query 1} \downarrow \quad ... \quad \downarrow \text{Query 5} \\ \left[\begin{array}{ccccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{array}\right] \end{array} \begin{array}{l} \leftarrow \text{Fragment 1} \\ \\ \\ \quad ... \\ \\ \leftarrow \text{Fragment 5} \end{array}$$

Now, suppose in a given epoch some node i receives a number of tuples corresponding to each fragment; e.g., it receives 23 tuples satisfying queries 1 and 3 only (row 1), 43 satisfying queries 2 and 3 only (row 2), etc. We can also represent this as a matrix called A_i:

$$A_i^{\mathsf{T}} = \left[\begin{array}{ccccc} 23 & 43 & 18 & 109 & 13 \end{array}\right]$$

Given the two matrices, we can now compute the local count for the first query (the first column of F) by summing the first and fourth entries in A_i, the second query by summing the second and fifth entries in A_i. In algebraic form $A_i^{\mathsf{T}} \times F$ will produce a one-row matrix with each column representing the count for the respective query. Encoding the information as matrix A_i is not more compact than sending the traditional set of five PSRs (one for each query). However, if we can find a reduced matrix A_i' – one with empty entries that do not need to be communicated – such that $A_i'^{\mathsf{T}} \times F = A_i^{\mathsf{T}} \times F$, we can save communication at the expense of more computation.

This is indeed possible in our example. First, note that fragment 4 is the OR of the *non-overlapping* fragments 1 and 3 (i.e. their conjunction equals zero). Now, observe the significance of that fact with respect to computing our COUNT queries: when summing up the counts for those queries that correspond to fragment 1 (queries 1 and 3), we can ignore the count of fragment 3 since its entries for those queries are zero. Similarly, when summing up the counts for queries overlapping fragment 3 (query 4), we can ignore the count of fragment 1. Because of this property, we can add the count associated with fragment 4 into *both* of the counts for fragments 1 and 3 without double-counting in the final answer, as follows:

$$A'^{\mathsf{T}} = \left[\begin{array}{ccccc} 23{+}109{=}132 & 43 & 18{+}109{=}127 & 109{\rightarrow}\emptyset & 13 \end{array}\right]$$

Using this new A_i', $A_i'^{\mathsf{T}} \times F$ will still produce the correct answer for each query, even though A' has more empty entries. And since A_i' has an empty entry, there is a corresponding savings in-network bandwidth, sending only four PSRs instead of five. In essence, we only need to execute four queries instead of the original five. The key observation is that the size of A_i' is equal to the number of *independent rows* in F, or the *rank* of F; the exact definition of independence depends on the aggregation function as we discuss next. In all cases the rank of F will always be less than or equal to $min(f, q)$. Therefore we will never need more than q PSRs, which is no worse than the no-sharing scenario.

5.1.2 Taxonomy Of Aggregates

The optimization presented in the previous section (based on ORing non-overlapping fragments) works for all distributive and algebraic aggregate functions (see Section 4.2 for types of aggregation functions). However, some distributive and algebraic aggregate functions have special properties that allow more powerful solutions to be used that exploit additional sharing opportunities. We categorize these aggregates into three broad categories: *linear*, *duplicate insensitive*, and *general*. These three categories map to different variations of the problem and require separate solutions. We first discuss the taxonomy and then briefly introduce our solutions.

Formally, we use the term *linear* for aggregate functions whose fragment matrix entries form a field (in the algebraic sense) under two operations, one used for combining rows, the other for scaling rows by constants. An important necessary property of a field is that there be *inverses* for all values under both operators. Among the familiar SQL aggregates, note that there is no natural inverse for MIN and MAX under the natural combination operator: given that $z = \text{MAX}(x, y)$, there is no unique y^{-1} such that $\text{MAX}(z, y^{-1}) = x$. Hence these are not linear. Another category we consider are *duplicate insensitive* aggregates, which produce the same result regardless of the number of occurrences of a specific datum. The table below lists a few example aggregate functions for each category:

	Non-linear	Linear
Duplicate Sensitive	k-MAX, k-MIN	SUM, COUNT, AVERAGE
Duplicate Insensitive	MIN, MAX, BLOOM FILTER, logical AND/OR	Spectral Bloom filters [15], Set expressions with updates [23]

The intuition for why k-MAX and k-MIN (the multi-set of the top k highest/lowest datums) are non-linear is analogous to that of MAX and MIN. k-MAX/MIN are also duplicate sensitive since evaluating each additional copy of the same highest datum would expel the kth highest datum due to the multi-set semantics.

Spectral Bloom filters are an extension of Bloom filters that keep a frequency associated with each bit. The frequency is incremented when a datum maps to that bit, and can be decremented when a datum is removed from the filter. This is linear because the frequencies can be added/subtracted to each other and can be scaled by a real number. In addition the output of the filter is based on whether the frequency is greater than zero or not, so counting the same datum twice may produce an inflated frequency value but does not change the output.

In Section 5.4 we address linear aggregates where this problem can be reduced directly to rank-revealing linear algebra factorization of matrix F, and polynomial-time techniques from the literature directly lead us to an efficient solution. For duplicate insensitive aggregates, we explain in Section 5.5 that the problem is a known NP-Hard problem and has higher computational complexity; in these cases we develop a family of heuristics that we evaluate experimentally. Finally for aggregates that are neither linear nor duplicate insensitive, the most conservative optimization algorithm must be used. We stress that for both linear and

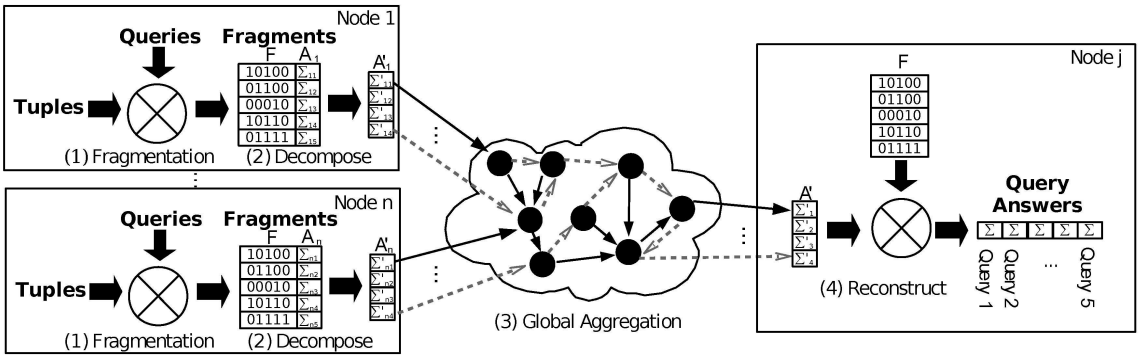


Figure 5.1: Tuples are first aggregated by fragment (1) into a local A_i PSR. F and A_i are then decomposed (2) to form A'_i. Each entry in A'_i is then aggregated over all nodes (3) in separate aggregate trees. The final global value for each entry in A' sent to some node j. Node j can then reconstruct (4) the answers to every query and distribute the result.

duplicate insensitive aggregates, our solutions will never require more global aggregate computations than the no-sharing scenario.

We now discuss the architecture and the general solution to this problem.

5.2 Architecture

The general technique for performing our multi-query optimization has four phases. First at each node, i, we need to create the initial F and A_i matrices in the *fragmentation* phase. Second, we can *decompose* F and A_i into a smaller A'_i. Third, we perform the *global aggregation* of all local A'_i's across all nodes. Finally, we can *reconstruct* the final answers to each query at some node j. This process is illustrated in Figure 5.1 and described in detail below.

In the first phase, fragmentation, we are using the same technique presented in [44]. Each tuple is locally evaluated against each query's predicates to determine on-the-fly which fragment the tuple belongs to. We can use techniques such as group filters [55] to efficiently evaluate the predicates. Once the fragment is determined, the tuple is added to the fragment's corresponding local PSR in A_i.

In the second phase, decomposition, each node will locally apply the decomposition algorithm to F and A_i to produce a smaller matrix, A'_i. The specific decomposition algorithm used is dependent on the type of aggregate function being computed. In Section 5.3 we present the basic algorithm that applies to all functions. Section 5.4 shows an algorithm that can be used for linear aggregate functions, and, in Section 5.5 we show a family of heuristic algorithms that work for duplicate insensitive functions.

We require that every node in the system use the same F matrix for decomposition. The F matrices must be the same so that every entry in A'_i has the same meaning, or in other words, contains a piece of the answer to same set of queries. Nodes that do not have any tuples for a particular fragment will have an empty PSR in A_i. In Section 5.6.1, we explain how to synchronize F on all nodes as data is changing locally; for

duplicate insensitive aggregate functions, we are able to *eliminate this requirement altogether*.

In the third phase, global aggregation, we aggregate each of the A'_i's over all nodes in the system to produce the global A'. Since we want to maintain the load balanced property of the non-sharing case, we aggregate each entry/fragment in A' separately in its own aggregation tree. Once the final value has been computed for an entry of A' at the root of its respective aggregation tree, the PSR is sent to a single coordinator node for reconstruction.

The fourth phase, reconstruction, begins once the coordinator node has received each of the globally computed A' entries. Using the F matrix (or its decomposition) the answer to all queries can be computed. The reconstruction algorithm is related to the specific deconstruction algorithm used, and is also described in the respective sections.

We take a moment to highlight the basic costs and benefits of this method. Both the sharing and no-sharing methods must disseminate every query to all nodes. This cost is same for both methods and is amortized over the life of the continuous query. Our method introduces the cost of having all nodes agree on the same binary F matrix, the cost to collect all of the A' entries on a single node, and, finally the cost to disseminate the answer to each node that issued the query. The benefit is derived from executing fewer global aggregations (in the third phase). The degree of benefit is dependent on the data/query workload. In Section 5.7 we analytically show for which range of scenarios this method is beneficial.

5.3 Basic Decomposition Solution

Our first algorithm, basic decomposition, applies to all aggregation functions, and directly follows the intuition behind the optimization we presented in the previous section. Our aim is to find the smallest set of basis rows, such that each row is exactly the disjunction of two or more basis rows that are non-overlapping – i.e., their conjunction is empty. If the basis rows were to overlap, then a tuple would be aggregated multiple times for the same query.

Formally, we want to find the basis rows in F under a limited algebra. Standard boolean logic does not allow us to express the requirement that basis rows be non-overlapping. Instead, we can define an algebra using a 3-valued logic (with values of 0, 1, and I for "invalid") and a single binary operator called ONCE. The output of ONCE is 1 if and only if exactly one input is 1. If both inputs are 0, the output of ONCE is 0, and if both inputs are 1 the output is I. Using this algebra, the minimal set of rows which can be ONCEed to form every row in F is the minimal basis set, and our target solution. The I value is used to prevent any tuple from being counted more than once for the same query.

The exhaustive search solution is prohibitively expensive, since if each row is q bits there are 2^{2^q} possible solutions. While this search space can be aggressively pruned, it is still too large. Even a greedy heuristic is very expensive computationally, since there is a total of 2^q choices (the number of possible rows) at each step – simply enumerating this list to find the locally optimal choice is clearly impractical.

To approach this problem, we introduce a simple heuristic that attempts to find basis rows using the

existing rows in F. Given two rows, i and j, if j is a subset of i then j is covering those bits in i that they have in common. We can therefore decompose i to remove those bits that are in common. When we do that, we need to alter A by adding the PSR from i's entry to j's entry.

We can define a DECOMPOSE operation as:

DECOMPOSE(F, A_i, i, j):
  if ($i \neq j$) AND ($\neg F[i] \& F[j] = 0$) **then** \\ *ONCE($F[i], F[j]$)*
    $F[i] = F[i]$ XOR $F[j]$
    $A_i[j] = A[j] + A[i]$
  else return invalid

A simple algorithm can iteratively apply DECOMPOSE until no more valid operations can be found. This decomposition algorithm, will transform F and A_i into F' and A'_i:

BASIC DECOMPOSITION(F, A_i):
  boolean progress = true
  while progress = true
    progress = false
    for all rows $i \in F$
      for all rows $j \in F$
        if Decompose(F, A_i, i, j) $\neq$ invalid
          then progress = true
  for all rows $k \in A_i$
    if $|F[k]| = 0$ **then**
      $A_i[k] = \emptyset$ \\ *rows in F with all 0's*

Reconstruction is straightforward since $A_i'^{\mathsf{T}} \times F' = A_i^{\mathsf{T}} \times F$.

The running time of the basic decomposition algorithm is $O(f^3)$, where f is the number of rows in F. Since the basic decomposition is searching a small portion of the search space, it is not expected to produce the smallest basis set. Furthermore, it is the only algorithm we present that can produce an answer worse than no-sharing. The algorithm starts with f basis rows, where f can be greater than q, and attempts to reduce the size of this initial basis. This reduction may not always be sufficient to find a basis that is smaller than or equal to q (although one such basis must exist). In these cases we revert to a $q \times q$ identity matrix which is equivalent to a no-sharing solution. However, this simple algorithm does provide a foundation for our other solutions.

5.4 Linear Aggregate Functions

If the aggregate function is linear, such as COUNT, SUM, or AVERAGE, we are no longer constrained to using the limited algebra from the previous section. Instead, we can treat the matrix entries as real

numbers and use linear algebra techniques akin to Gaussian Elimination, adding and subtracting rows in F from each other, and multiplying these rows by scalars. Our goal of reducing the size of A_i can therefore be accomplished by finding the minimal set of linearly independent rows F' in F, or the rank of F. By definition F can be reconstructed from F', so we can create A'_i from A_i at the same time and still correctly answer every query during the reconstruction phase.

For example, suppose we are calculating the COUNT for these five queries with this F and A_i matrix:

$$F = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} A_i = \begin{bmatrix} 13 \\ 54 \\ 24 \\ 78 \\ 32 \end{bmatrix}$$

The answer to the first query (in the leftmost column) is $13 + 54 + 78 + 32$ or 177. The complete solution matrix can be computed using $A_i^{\mathsf{T}} \times F$.

It turns out that we can express F and A_i using only four rows:

$$F' = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} A'_i = \begin{bmatrix} 177 \\ -30 \\ 37 \\ 13 \end{bmatrix}$$

Using F' and A'_i we can still produce the correct solution matrix, using $A_i'^{\mathsf{T}} \times F'$. In this example we used Gaussian Elimination on F to find the smallest set of basis rows. We will now discuss how to solve this problem using more efficient algorithms.

In numerical computing, *rank-revealing factorizations* are used to the find the minimal set of basis rows. We will apply three well-studied factorizations to our problem: the LU, QR, and SVD decompositions. These algorithms will decompose F into two or more matrices that can be used in local decomposition to transform A_i into A'_i and then to reconstruct A' into the query answers at the coordinator node. These factorization methods and their implementations are well studied in the numerical computing literature [6]. We now present formulations for utilizing these factoring methods.

An LU algorithm factors F into a lower triangular matrix L and an upper triangular matrix U such that $L \times U = F$. In the decomposition phase we can form A'_i using $A_i^{\mathsf{T}} \times L$ and remove any entries in A'_i whose corresponding row in U is composed of all zeros. Reconstruction at the coordinator is simply $A' \times U$. We can safely remove the entries in A'_i whose corresponding row in L is all zeros because in reconstruction those entries will always be multiplied be zero and thus do not contribute to any results. During reconstruction we insert null entries in A' as placeholders to insure the size of A' is correct for the matrix multiplication.

Using QR factoring is very similar to using LU. In this case, the QR algorithm factors F into a general matrix Q and an upper triangular matrix R such that $Q \times R = F$. We form A'_i using $A_i^{\mathsf{T}} \times Q$

and remove any entries in A'_i whose corresponding row in R is composed of all zeros. Reconstruction is accomplished using $A' \times R$.

SVD factors F into three matrices, U, S, and V^T. A'_i is formed in decomposition using $A_i^\mathsf{T} \times U \times S$. Using this method, we remove entries from A'_i whose corresponding row in S is zero. Reconstruction is accomplished by computing the product of A' and V^T. With all three algorithms, the factorization of F is deterministic and therefore the same on all nodes, allowing us to aggregate A'_is from all nodes before performing reconstruction.

These algorithms all have a running time of $O(m \times n^2)$ where m is the size of the smaller dimension of F and n is the larger dimension [6]. In addition, all three methods would be optimal (finding the smallest basis set and thus reducing F and A_i to the smallest possible sizes) using infinite precision floating point math. However, in practice these are computed on finite-precision computers which commonly use 64 bits to represent a floating point number. Factorization requires performing many floating point multiplications and divisions which may create rounding errors that are further exacerbated through additional operations. While LU factorization is especially prone to the finite precision problem, QR factoring is less so, and SVD is the least likely to produce sub-optimal reductions in A''s size. Due to this practical limitation, the factorization may not reach the optimal size. In no case will any of these algorithms produce an answer that requires more global aggregations than the no-sharing scenario. In addition, these rounding error may introduce errors in A' and therefore perturb the query results. However, these algorithms, in particular SVD, are considered robust and used in many applications.

5.5 Duplicate Insensitive Aggregate Functions

The previous algorithms preserve the invariant that each tuple that satisfies a particular query will be aggregated exactly once for that query. However, some aggregate functions, such as MIN and MAX, will still produce the same answer even if a tuple is aggregated more than once. We can take advantage of this property when decomposing F and achieve a higher communication savings compared to the previous algorithms. Consider this simple example:

$$F = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix} \quad F' = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

We notice that the fifth row of F is equal to the OR of the second and third (or second and fourth, or third and fourth). Thus we can define a matrix F' that removes this redundant row. The corresponding operation to the A matrix is to aggregate the fifth entry with the second entry, aggregate the fifth entry with the third entry, and then remove the fifth entry. Intuitively, this is moving the data from the fifth fragment to both the second and third fragments.

Similar to the previous sections, the goal is to find the minimum number of *independent rows*. But in this case, the independent rows are selected such that all rows in F can be obtained by combining rows in the basis using only the standard OR operation, rather than ONCE.

This problem is also known as the *set basis* or *boolean basis* problem. The problem can be described succinctly as follows. Given a collection of sets $S = \{S_1, S_2, ...S_s\}$, a basis B is defined as a collection of sets such that for each $S_i \in S$ there exists a subset of B whose union equals S_i; the set basis problem is to find the smallest such basis set. Our problem is the same, where S = rows of F and B = rows of F'. The set of possible basis sets is 2^{2^n} where n is the number of elements in $\bigcup S$. This problem was proved NP-Hard by Stockmeyer [73], and was later shown to be inapproximable to within any constant factor [52]. To our knowledge, ours is the first heuristic approximation algorithm for the general problem. In [51] Lubiw shows that the problem can be solved for some limited classes of F matrices, but these do not apply in our domain.

As with the general decomposition problem in Section 5.3, the search space of our set basis problem is severely exponential in q. To avoid exhaustive enumeration, our approach for finding the minimal basis set, F', is to start with the simplest basis set, a $q \times q$ identity matrix (which is equivalent to executing each query independently), and apply transformations. The most intuitive transformation is to OR two existing rows in F', i and j, to create a third row k. Using this transformation (and the ability to remove rows from F' one could exhaustively search for the minimal basis set. This approach is obviously not feasible.

We apply two constraints to the exhaustive method in order to make our approach feasible. First, after applying the OR transformation, at least one of the existing rows, i or j, is always immediately removed. This ensures that the size of the basis set never increases. Second, we maintain the invariant that after each transformation the set is still a valid basis of F.

We can now formally define two operations, BLEND and COLLAPSE which satisfy these invariants. Given a matrix F and a basis F' for F, both operations overwrite a row $F'[i]$ with the OR of row $F'[i]$ and another row $F'[j]$. COLLAPSE then removes row j from F', whereas BLEND leaves row j intact. After performing one of these operations, if the new F' still forms a basis for F then the operation is valid; otherwise the original F' is kept.

COLLAPSE is the operation that achieves a benefit, by reducing the size of the basis set by one. COLLAPSE is exploiting the co-occurrence of a bit pattern in F. However, it may not be valid to apply COLLAPSE until one or more BLEND operations are performed. The intuition for this is that when the bit pattern in some input row can be used in multiple basis rows, BLEND preserves the original row so that it can be used as, or part of, another basis row. Consider matrix F, and the following *invalid* COLLAPSE transformation:

$$F = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad F' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We cannot directly COLLAPSE rows one and four in F' as shown above. The resulting F' is no longer able to

reconstruct the first or fourth rows in F via any combination of ORs; we call such a transformation *invalid*. However, if we first BLEND rows two and four (leaving row four), we can then COLLAPSE rows one and four, as shown next:

$$F' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Using these two operations we can search a subset of the overall search space for the minimal basis set. A simple search algorithm, called BASIC COMPOSITION, performs BLEND or COLLAPSE in random order until no more operations can be executed. The pseudo-code is shown below:

```
BASIC COMPOSITION(F)
  F' = qxq identity matrix
  boolean progress = true
  while progress = true
    progress = false
    for all rows i ∈ F'
      for all rows j ∈ F'
        if i ≠ j then
          if COLLAPSE (F, F', i, j) ≠invalid then
            progress = true
            break to while loop
          if BLEND (F, F', i, j) ≠invalid then
            progress = true
            break to while loop
```

A'_i can be calculated by aggregating together each element in A_i that corresponds to a row in F which is equal to or a superset of the A'_i entry's corresponding F' row.

There are three key limitations of this algorithm:

- Once an operation is performed it can not be undone: both operations are non-invertible and there is no back-tracking. This limits the overall effectiveness of finding the minimal basis set since the algorithm can get trapped in local minima.
- The random order in which operations are performed can determine the quality of the local minimum found.
- At any given point there are $O(f^2)$ possible row combinations to choose from. Finding a valid COLLAPSE or BLEND is time consuming.

In effect, the algorithm takes a single random walk through the limited search space. For some workloads, the optimal solution may not even be attainable with this method. However, while this heuristic algorithm gives no guarantees on how small the basis set will be, it will never be worse than the no-sharing solution. We will show in Section 5.8 that this heuristic is often able to find 50% of the achievable reductions in the size of the basis set, but its running time is extremely long.

5.5.1 Refinements

Our first refinement takes a slightly different approach. Instead of optimizing every query at once, we incrementally add one query at time optimizing as we go. The two key observations are (1) that a valid covering for $q-1$ queries can cover q queries with the addition of a single row which only satisfies the new query and (2) the optimal solution for q queries given an optimal basis solution for $q-1$ queries and the single basis row for the qth query will only have up to one valid COLLAPSE operation.

Using these observations we can define the ADD COMPOSITION algorithm which incrementally optimizes queries one at a time:

ADD COMPOSITION(F, F', $start$)
Require: F' has $start$ columns
let q = the number of queries \\ *columns in* F
let f = the number of rows \\ *rows in* F'
for $c = start + 1$ **up to** q
Expand F' to $(f+1) \times c$ with 0's
$F'[f+1][c] = 1$
F_c = Project(F,c) \\ *See Following Algorithm*
Optimize(F_c,F',$f+1$)
return F'

PROJECT(S, $columns$)
for all rows $i \in S$
for all cols $j \in S$
if $j \leq columns$ **then**
$S''[i][j] = S[i][j]$
else $S''[i][j] = 0$
S' = unique rows in S''
return S'

The OPTIMIZE step in ADD COMPOSITION is very similar to the repeat loop in BASIC COMPOSITION. It has a *search loop* that continues looking for combinations of rows that can be used in a COLLAPSE or BLEND operation until there are no such combinations. OPTIMIZE has two key improvements over the BASIC COMPOSITION. First, COLLAPSEs and BLENDs are not considered if they combine two old (optimized)

rows. Second, since only one row was added to F', once a COLLAPSE is performed the optimization is over and the search loop is stopped, since no additional COLLAPSEs will be found. As shown in Section 5.8 this method is considerably faster and still equally effective at finding a small basis set compared to the BASIC COMPOSITION algorithm.

We consider three dimensions for search loop strategies:

- **Number of operations per iteration:**
  - *O:* Perform only one operation per search loop and then restart the loop from beginning.
  - *M:* Perform multiple operations per search loop only restarting after every combination of rows are tried.
- **Operation preference:**
  - *A:* Attempt COLLAPSE first, but if not valid attempt BLEND before continuing the search.
  - *R:* Perform all COLLAPSEs while searching, but delay any BLENDs found till the end of the search loop.
  - *S:* First search and perform only COLLAPSE operations, then search for and perform any BLENDs. This requires two passes over all row pairs per loop.
- **Operation timing:**
  - *W:* Execute operations immediately and consider the new row formed in the same loop.
  - *D:* Execute operations immediately but delay considering the new row for additional operations till the next loop.
  - *P:* Enqueue operations till the end of the search loop and then execute all operations.

The BASIC COMPOSITION algorithm shown uses the O/R strategy. The algorithm performs one operation per iteration of the outer loop. So after each operation, it will begin the search again from the beginning. The algorithm favors COLLAPSE by attempting that operation first. The operator timing dimension is not relevant for strategies that only perform one operation per iteration. Note that the BASIC COMPOSITION can be modified to use any of the possible search strategies. In the evaluation section we only show the strategy that performed the best in our experiments, M/A/W.

There are only twelve search strategies possible using the three dimensions evaluated (when performing only one operation per search loop, operation timing is not relevant). All twelve are experimentally evaluated in Section 5.8.

5.6 Practical Matters

In this section we discuss how we ensure that every node has the correct F matrix, whether through explicit or implicit communication, and its associated network overhead. We then discuss how to extend our

methods to work for a larger class of complex queries.

5.6.1 Synchronizing F Across the Network

In order to ensure the PSRs in A' that are communicated from one node to another are correctly decoded we must guarantee that every node has the same F matrix. Otherwise, during the global aggregation or reconstruction phases, the PSRs in A' may be incorrectly aggregated causing the query results to be wrong. This is very important for correctness of some decomposition algorithms such as the linear algebra routines LU, QR, and SVD. For the other decomposition algorithms presented there is an optimization to the architecture that eliminates this requirement. We first describe a simple method for ensuring all nodes have the same F and then describe the optimization.

At the end of every aggregation epoch (after the node has collected all the raw data necessary to compute the aggregates for that epoch) each node, i, computes its local F matrix, F_i. Since each node may have a different distribution of data, the matrix F_i at node i may differ from matrix F_j at node $j \neq i$. The global F is the set union of the rows in all local F_i's.

This can be computed like any other aggregate using a tree. All the leaves of the tree send their complete F_i to their parents. Their parents compute the union over all their children, and send the result to their parent. At the root of this aggregation tree, the global F is computed. The global F is then multicast to every node on the reverse path of the aggregation tree.

For subsequent epochs only additions to F need to be transmitted up or down the aggregation tree. Deletions can also be propagated up the aggregation tree, however any node along the path can stop the propagation (which prevents a change to the global F) if it has at least one other child (or itself) still needing that row. The addition or deletion of a query will also change F. Query (column) deletions require no communication (every node simply removes the column for F). The addition of a query (column) affects every row in F, but in a limited fashion. Each row is either extended with a 1, a 0, or both (which requires duplicating the old row). This can be compactly transmitted as a modification bitmap with two bits per existing row. The global modification bitmap is the OR of every node's individual modification bitmap which can also be efficiently computed as an aggregate.

Once all nodes have the global F, the general computation of the query aggregates can begin. This synchronization method has the negative effect of delaying all results for at least the duration of one global aggregation plus one global multicast. In practice, the actual delay must be sufficiently long to accommodate worst case delays in the network.

The exact communication cost of this method is dependent on the query/data workload. However, given a constant set of q queries and a set of n nodes, we can show the worst case cost of synchronizing F for each additional bitmap, and for how many epochs the system must remain unchanged to recoup the cost.

The worst case communication cost occurs if at least every leaf node in the aggregation tree requires the addition of the same new row in a given aggregation epoch. In this situation every node will need to transmit the new row in F up and down the aggregation tree which yields a cost of $2 \times n \times q$ bits per row. If

only one node requires the new row the cost is roughly $n \times q + \log(n) \times q$ as only one node is sending data up the aggregation tree.

Assume the size of each PSR is p bits. The savings realized from sharing will never be less than the eventual total gain, G_t. During each epoch, $(1 - G_t) \times q$ aggregates are being computed instead of q queries in the no-sharing scenario, for a benefit of $((q - (1 - G_t) \times q) \times p) \times n$ or $G_t \times q \times n \times p$ bits per epoch. We reach the break-even point after $\frac{2 \times n \times q}{G_t \times q \times n \times p} = \frac{2}{G_t \times p}$ epochs. If multiple rows must be added at the same time, the number of epochs till the break-even point increases proportionally.

The basic decomposition and the algorithms for duplicate insensitive aggregates do not require a global F and can avoid the associated costs. Instead, it is sufficient to annotate every entry in A' with its corresponding binary row in F'. Since every aggregation tree is required to have an identifier (such as a query identifier) to distinguish one tree from another, the basis row entry can be used as the identifier. This is possible since the reconstruction phase does not any require additional information about the decomposition.

While this optimization does not apply to linear aggregates there are other techniques that could be considered. For some query workloads a static analysis of the query predicates may be sufficient to compute a superset of F. This can be further extended to handle the actual data distribution by having nodes compactly communicate which portions of the data space they have. We leave a complete analysis of this optimization for future work.

5.6.2 Complex Queries

Our query workload to this point might seem limited: sets of continuous queries that are identical except for their selection predicates. In this section we observe that our techniques can be applied to richer mixes of continuous queries, as a complement to other multiquery optimization approaches.

For example, [46, 44] discuss optimizing sharing with queries that have different epoch parameters. Their methods partition the stream into smaller epochs that can later be combined to answer each of the queries. One can view the epoch-share optimization as query rewriting, producing a set of queries with the same epoch parameters, which are post-processed to properly answer each specific query. In that scenario, our technique is applied to the rewritten queries. Similarly, queries with different grouping attributes can also be optimized for sharing. In that case, the smallest groups being calculated would be treated as separate partitions of the data that are then optimized separately by our techniques. After processing the results can be rolled-up according to each queries specification.

Our approach does not depend on a uniform aggregation expression across queries. Queries that include multiple aggregate functions, or the same function over different attributes, or queries that require different aggregate functions can be optimized as one in our approach – as long as the same decomposition can be used for all the aggregate expressions. In these cases, the PSR contained in A or A' is the concatenation of each PSR needed to answer all aggregate functions. In those cases where different decompositions must be used (e.g., one function is a MAX and another is a COUNT) then they can be separately optimized and executed using our techniques.

Our results show that there is a clear choice of which optimization technique to use for most classes of aggregate functions. However, if a function is both linear and duplicate-insensitive, it is unclear which technique to apply. While few functions fall in this category (see Section 5.1.2), for those functions the selection of algorithm will be dependent on the specific workload. Characterizing the tradeoffs among workloads for these unusual functions remains an open problem.

5.7 Potential Gains

Before we evaluate the effectiveness of our techniques experimentally, we explore the analytical question of identifying query/data workloads that should lead to significant beneficial sharing, and quantifying that potential benefit. This will provide us a framework for evaluating how close our "optimization" techniques come to a true optimum. In this section, we show that there are cases where sharing leads to arbitrarily high gain, given a sufficient number of queries. We present two constructions, one designed for duplicate insensitive query workloads, the other for duplicate sensitive workloads. Our goal is to construct a workload that maximizes the sharing or benefit potential. We define the total gain, G_t as:

$$G_t = 1 - (\text{\# aggregates executed} \div \text{\# queries answered})$$

We also define the fragment gain which is the gain over computing each fragment, s as:

$$G_f = 1 - (\text{\# aggregates executed} \div \text{\# fragments})$$

The total gain, G_t, is the most important metric, since an effective decomposition algorithm can translate this sharing potential into a proportional amount of network bandwidth savings. The fragment gain, G_f, is the benefit over computing every fragment in F.

5.7.1 Duplicate Insensitive

To maximize the sharing potential we start with b base queries $(b_1, b_2, b_3, ...b_b)$ and data that satisfies every conjunctive combination of the b queries $(\{b_1\}, \{b_2\}, \{b_3\}, ...\{b_1, b_2\},$
$\{b_1, b_3\}, ...\{b_1, b_2, b_3, ...b_b\})$ such that we have $2^b - 1$ fragments (the -1 is for data that satisfies no queries). At this stage, no sharing is beneficial since only b aggregates are actually needed (one for each query).

Using the initial b queries, we can write an additional $2^b - 1 - b$ queries by combining them via *disjunction*, i.e. query x matches data that satisfies query b_1 and b_2, query y matches data satisfying b_2 or b_3, etc. One such additional query is outlined in Figure 5.2(a). In this case there are 2^b such combinations from which we subtract the original b queries and the combination that is the disjunction of the empty set of queries. The additional queries do not introduce any additional fragments.

These new $2^b - 1 - b$ queries can be answered if we have the answers to the original b queries. Since the aggregate functions we consider here are duplicate insensitive, the disjunction of multiple queries is simply their aggregation. So if we compute the aggregates for the original b queries, we can clearly answer

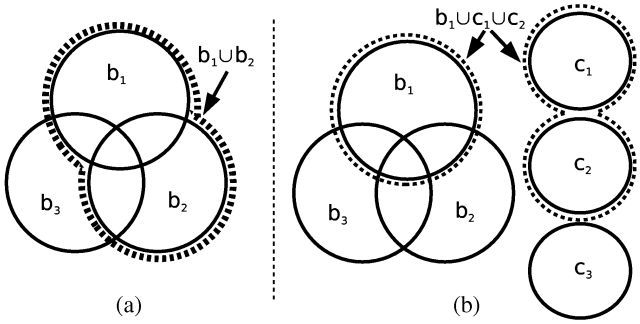


Figure 5.2: Example Venn diagrams for duplicate insensitive construction (a) and the duplicate sensitive construction (b). In (a) the additional query $b_1 \cup b_2$ is outlined. In (b) the additional query $b_1 \cup c_1 \cup c_2$ is outlined.

the original b queries plus the new $2^b - b - 1$ queries for a total of $2^b - 1$ queries. Thus, $G_t = G_f = 1 - \frac{b}{2^b - 1}$. As $b \to \infty$ the gain approaches 1 which is maximal.

The intuition behind this construction is that queries that are the disjunction of other queries lead to sharing opportunities. While the first b base queries have significant amounts of overlap, the overlap creates additional fragments because each query has a unique set of matching data. It should be noted that none of the $2^b - 1$ fragments created from the b base queries are actually used to answer any queries, instead the b base queries are computed directly and used to compute the additional $2^b - 1 - b$ queries. This is only possible because the aggregation functions are duplicate insensitive and the overlap in data between the b base queries does not affect the answer for the additional queries.

Furthermore, it is not necessary that the b base queries are explicitly requested by users. If only the additional queries were issued, those queries could still be answered using just b global fragments. This means that the gain is realized when the query set is just the disjunction of a smaller number of overlapping fragments.

5.7.2 Duplicate Sensitive

This construction is similar to the previous construction, with b base queries and $2^b - 1$ fragments. Now we add c non-overlapping queries such that data that satisfies one of the c queries and does not match any other query (from b or c). Thus, there are c additional fragments for a total of $b + c$ fragments.

We now add $2^c - 1 - c$ additional queries based solely on the c non-overlapping queries by taking the disjunction of every possible combination of the c queries. These queries can be answered by aggregating the answers from the original c queries. Note, this does not count any tuple twice since the c queries were non-overlapping.

Finally, we add $(2^c - 1) \times (b)$ more queries by taking the disjunction of every possible combination of the c queries and exactly one query from the b base queries. For example, we take $c_1 \cup b_1$, $c_2 \cup b_2$, $c_1 \cup c_2 \cup b_1$ and $c_1 \cup c_2 \cup b_2$, etc. One such additional query is outlined in Figure 5.2(b). Since each of these additional queries is only the disjunction of one query from b, there is still no overlap, so no data is counted

multiple times.

In summary we have $b + c + 2^c - 1 - c + (2^c - 1) \times b$ queries which could be answered using $b + c$ fragments. This leads to a total gain of $1 - \frac{b+c}{2^c-1+b\times 2^c}$, and fragment gain of $1 - \frac{b+c}{2^b-1+c}$. As b and c approach infinity, the total and fragment gains approach 1 which is maximal.

Intuitively, the c queries are the source of sharing, since we are able to construct many additional queries that are the disjunction of multiple base c queries. The b queries are the source of the fragment gain, since the overlap they create increases the number of fragments that are not needed.

5.8 Experimental Evaluation

In this section we evaluate the performance of the various decomposition methods we have presented. Rather than focus on a specific workload from a speculative application, we pursue an experimental methodology that allows us to map out a range of possible workloads, and we evaluate our techniques across that range.

We present a random workload generator based on our analysis of the gain potential in the previous section. This generator allows us to methodically vary the key parameters of interest in evaluating our techniques: the workload size, and the degree of *potential* benefit that our techniques can achieve. Within various settings of these parameters, we then compare the relative costs and benefits of our different techniques. After describing our workload generator, we present our experimental setup and our results.

5.8.1 Workload Generators

We designed a workload generator that allows us to input the *desired size* and *total gain* for a test F matrix. By controlling the total gain we are able to test the effectiveness of our algorithms. Using the combination of the two knobs we can explore various workloads. We have two generators, one for duplicate sensitive aggregates and one for duplicate insensitive aggregates, that create test F matrices. The constructions from the previous section are used to develop these generators.

For the duplicate insensitive generator we can calculate the number of basis rows, b, the number of fragments, f, and the number of queries, q, based on the desired matrix size and gain. Each of the b basis rows maps to one of the b base queries in the constructor. Instead of generating all $2^b - 1$ fragments, we uniformly at randomly select the f fragments from the set of possible fragments. Analogously, we uniformly at random select unique additional columns (queries) from the set of up to $2^b - b - 1$ possible additional queries. The generation is finalized by randomly permutating the order of the rows and columns.

This construction gives us a guarantee on the upper bound for the minimum number of basis rows needed, b. The optimal answer may in fact be smaller if the rows selected from the set of $2^b - 1$ can be further reduced. Since the rows are chosen randomly, such a reduction is unlikely. In our experiments, we attempt to check for any reduction using the most effective algorithms we have.

The duplicate sensitive generator works much the same, except with the addition of the c basis

rows. The additional columns (queries) are generated by ORing a random combination of the c base queries and up to one of the b base queries. Values for the number of b and c queries are randomly chosen such that their sum is the desired number of basis rows and such that b is large enough to ensure enough bitmaps can be generated and c is large enough that enough combination of queries can be generated.

Also note that the original b (and c) queries remain in the test matrix for both generators; while this may introduce a bias in the test, we are unable to remove these queries and still provide a reasonable bound on the optimal answer. Without knowing the optimal answer it is hard to judge the effectiveness of our algorithms.

5.8.2 Experimental Setup

We have implemented in Java all of the decomposition algorithms presented in the previous sections. Our experiments were run on dual 3.06GHz Pentium 4 Xeon (533Mhz FSB) machines with 2GB of RAM using the Sun Java JVM 1.5.06 on Linux. While our code makes no specific attempt to utilize the dual CPUs, the JVM may run the garbage collector and other maintenance tasks on the second CPU. All new JVM instances are first primed with a small matrix prior to any timing to allow the JVM to load and compile the class files.

Furthermore, we have also implemented our techniques on top of PIER. Three operators, a fragmentation, deconstruction, and reconstruction operators were added. This has enabled us to verify the benefits of our approach in a realistic setting, over a large-scale distributed query processing engine.

For the LU/QR/SVD decompositions we utilize the JLAPACK library, which is an automatic translation of the highly optimized Fortran 77 LAPACK 2.0 library. We also tested calling the Fortran library from C code. Our results showed that the Java version was about the same speed for the SVD routines (in fact slightly faster in some instances) while the more optimized LU and QR routines were about twice as slow on Java. Overall, the runtime differences are minor and do not effect our conclusions on relative speed or effectiveness so we only present the results from the Java version.

We employ three key metrics in our study:

- the *relative effectiveness* (which is equivalent to the relative decrease in-network bandwidth used for computing the aggregates)
- the *running times* of the decomposition routine
- the *absolute size* of the resulting matrix A' which is directly proportional to the network bandwidth

In particular, the relative effectiveness is based on the resulting size of A', the estimated optimal answer k and the number of queries q. It is defined as $(q - |A'|) \div (q - k)$ or the ratio of attained improvement to that of an optimal algorithm.

We vary the ratio of the number of fragments to queries (whether the test matrix is short, square, or long) from $1/2$ to 2. We repeat each experiment 10 times with different test matrices of the same charac-

teristics (size and total gain); the results presented include the average and plus/minus one standard deviation using error-bars.

5.8.3 Results

We first present the result from the linear decomposition algorithms, which prove effective and fast for linear aggregates. Next, we show results for the duplicate insensitive heuristics where we highlight the best and worst performing variants. Finally we discuss the results for the basic decomposition algorithm.

Linear Aggregates

Our first set of experiments evaluate the linear aggregate decompositions using the duplicate sensitive workload generator. In Section 5.4 we noted that LU, QR, and SVD can be used to compute to A'. They differ in their running times, and in practice there is a concern that numerical instability (via rounding during repeated floating-point arithmetic) can cause the techniques to incorrectly solve for the basis, and produce an inefficient F'. Figure 5.3 shows the resulting size of A', the overall effectiveness, and the running time for the three algorithms using square matrices (500 queries with 500 fragments).

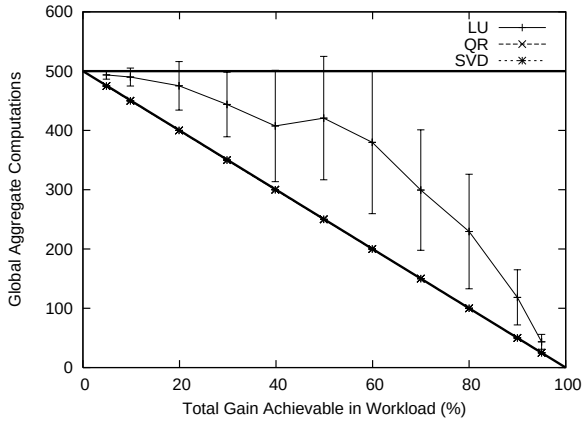
QR and SVD give always optimal results by finding the lowest rank and therefore the smallest A'. LU lagged in effectiveness due to sensitivity to precision limitations, with low effectiveness for matrices that had small potential gain and near optimal effectiveness for matrices that had high potential. As expected, LU and QR are substantially faster than SVD in our measurements by about an order of magnitude. Figure 5.4 shows that runtime increases polynomially ($O(q^3)$) as the size of the test matrix is increased.

F Size	Algorithm	A' size	% Effective	Runtime (ms)
250x500	LU	346.6 (164.1)	53.3%	884.8 (205.0)
250x500	QR	250.0 (151.8)	100%	1057.6 (24.1)
250x500	SVD	250.0 (151.8)	100%	9989.5 (545.5)
500x500	LU	345.6 (164.0)	48.5%	689.3 (159.6)
500x500	QR	250.0 (151.82)	100%	702.15 (81.48)
500x500	SVD	250.0 (151.82)	100%	6931.0 (431.0)
750x500	LU	235.6 (35.4)	83.4%	577.5 (306.4)
750x500	QR	175.0 (56.6)	100%	452.7 (81.0)
750x500	SVD	175.0 (56.6)	100%	3170.5 (311.9)

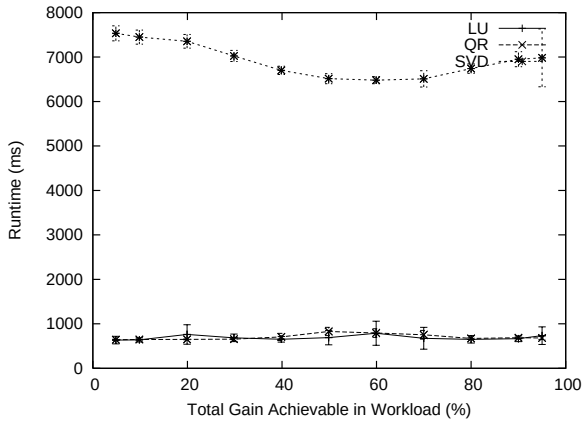
Table 5.1: Data for all linear algorithms averaged over all total gains for each particular matrix size. Standard deviation shown in parentheses.

In general, we found that the trends remain the same when the shape of the test matrix is changed. These results are summarized in the Table 5.1. QR and SVD remain optimal and LU has an overall effectiveness ranging from 50-85%. As expected, the running times increase for matrices with additional rows and decrease for matrices with fewer rows.

In summary, QR achieves the best tradeoff of effectiveness and speed. While SVD has been designed to be more robust to floating point precision limits, QR was able to perform just as well on this type



(a)



(b)

Figure 5.3: For 500x500 test matrices: (a) shows the resulting size of A'. The solid line at y=500 represents the starting size and the lower solid line represents optimal. QR and SVD are always optimal and precisely overlap each other and the lower solid line. (b) shows the running time for each.

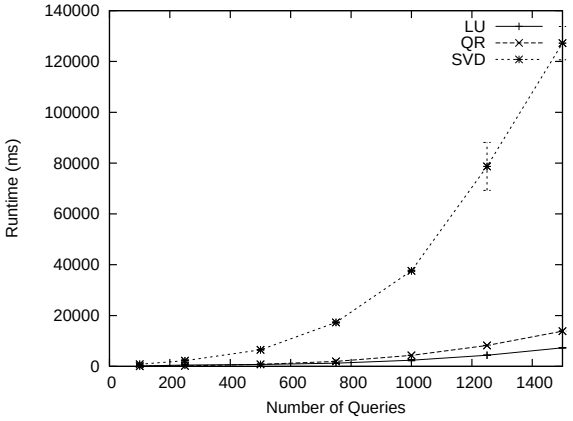


Figure 5.4: The running time as the size of square matrices are increased.

of binary matrix. LU has no benefit, since it is just as fast as QR, but not nearly as effective in finding the lowest rank.

Duplicate Insensitive Aggregates

Our second set of tests evaluate the composition family of heuristics using the duplicate insensitive workload generator. In Figure 5.5 we show the results. For clarity, we include a representative selection of the algorithms, including the BASIC COMPOSITION and the ADD COMPOSITION using five strategies (O/R, M/A/P, M/A/D, M/A/W, and M/R/W). The strategies for ADD COMPOSITION were chosen since they include: (1) the best strategy when used with BASIC COMPOSITION, (2) the worse performing, (3) the best performing and (4) two strategies similar in technique to the best performing. The strategies not shown have similar shaped curves falling somewhere in the spectrum outlined by those shown. The results for all algorithms are summarized in the Table 5.2 and have similar shaped curves falling somewhere in the spectrum outlined by those shown.

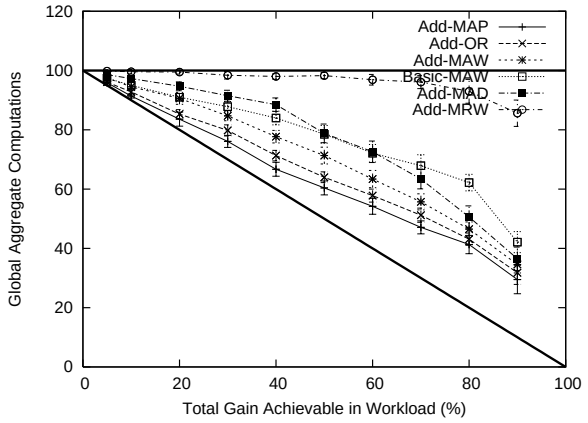
The results show that ADD COMPOSITION with the M/A/P search strategy is both the most effective and fastest, although not much more effective than with the O/R strategy (which is substantially slower). This is somewhat surprising given how different the M/A/P and O/R strategies seem. Also note that in most cases the relative effectiveness and the running time are inversely correlated. This indicates that some algorithms spend a lot of time searching and producing little benefit.

As explained in Section 5.5 the gain is revealed through the COLLAPSE operation. However, before COLLAPSE can be performed often a number of BLEND operations are needed before a COLLAPSE can be used. Search strategies that search for both COLLAPSE and BLEND at the same time tend to do better than strategies that search for more and more BLENDs after each other.

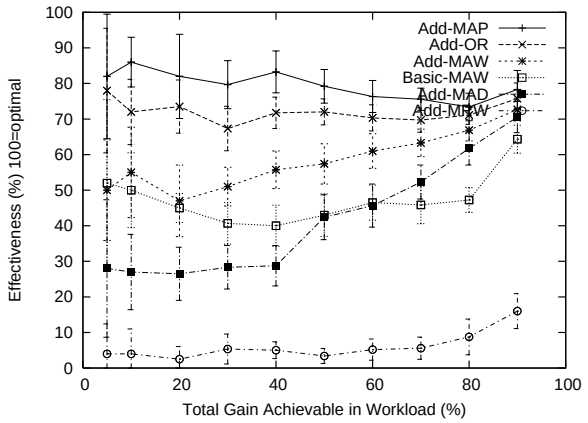
For example the M/S/W search strategy will first search for any possible COLLAPSE operations, and then search for BLEND operations separately. As an operation is performed the resulting row is considered for further operations in the same search loop. Even though COLLAPSEs are performed before BLENDs, once the strategy begins performing BLENDs it will continue to exclusively perform them until no more can be found. As a result, it gets stuck in this phase of the search loop. Even worse, it performs so many BLEND operations that they block future COLLAPSE operations and find a poor local minimum. This strategy often finds a local minimum and ends after it executes only two or three search loops.

In contrast, the O/R and M/A/P strategies are quick to search for more COLLAPSE operations after performing any operation. In the case of M/A/P, all possible operations with the given set of rows is computed, and they are then executed without further searching. While this tends to need many search loops, the strategy will not get caught in a long stretch of BLENDs. In the case of O/R, after every operation the search loop ends, and the search restarts. This strategy prevents getting stuck in the BLEND phase, but also wastes time continually searching the same portion of the search space over and over again after each operation. This causes the OR strategy to be considerably slower than the M/A/P strategy.

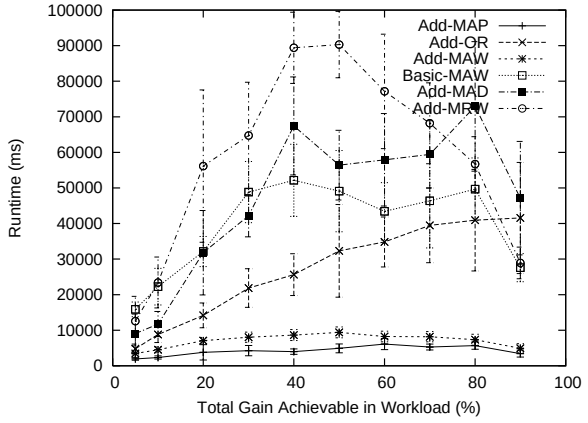
Figure 5.6 shows the running times of the fastest BASIC COMPOSITION and ADD COMPOSITION



(a)



(b)



(c)

Figure 5.5: For 100x100 test matrices (a) shows the resulting size of A'. The solid line at y=100 represents the starting size and the lower solid line represents optimal. (b) shows the relative effectiveness. (c) shows the running time for each.

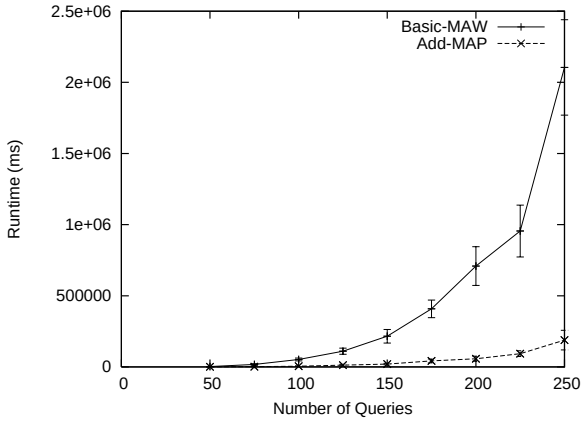


Figure 5.6: The running time as the size of a square matrix is increased.

strategies, for various sized square matrices. Unfortunately, none of the algorithms scale well as the matrix size is increased. However, the ADD COMPOSITION scales considerably better than the BASIC COMPOSITION algorithm. Effectiveness, not shown, remains the same or slightly increases as the size of the matrix increases.

Note that the super-linear scaling is not unexpected. The problem of finding a minimal boolean basis has been shown to be equivalent to finding the minimal number of maximal cliques in a bipartite graph [51]. Finding the maximal bi-clique is not only NP-Hard, but also is known to be hard to approximate. Our solution, while not fully exhaustive, considers a very large number of possibilities and produces near-optimal answers in our experiments.

In summary, the ADD COMPOSITION algorithm with the M/A/P search strategy is the clearly the winner. It is 70-90% effective in finding the smallest basis set, and is often the fastest algorithm for duplicate insensitive aggregates.

Basic Decomposition

As expected, our basic decomposition presented in Section 5.3 which works for all aggregate functions, is ineffective in most situations. Due to space limitations we do not show any results and only summarize our findings. For duplicate sensitive tests, the algorithm can often produce answers that are worse than no-sharing, generating an A' that has more entries than the number of queries, and in rare cases showing modest sharing of 5-20% optimal. The algorithm performs best in cases where there is large sharing potential. Perhaps the only redeeming characteristic of the algorithm is that it is fast, running faster than a half second for 500 queries, and only a few seconds for 1500 queries. For duplicate insensitive tests, the algorithm finds sharing potential extremely rarely, but runtime remains the same. This is expected, since the algorithm makes no attempt to exploit the duplicate insensitive property.

Given these results, it is clear that this general-purpose technique should not be used when our

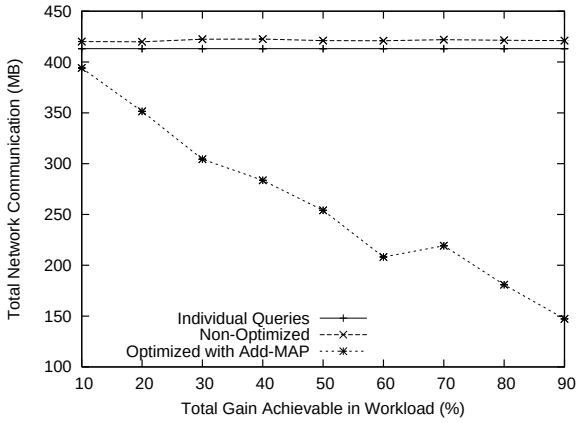


Figure 5.7: Total network communication savings in PIER.

special-purpose solutions can be used (i.e., for duplicate insensitive and linear aggregates.) Non-linear, duplicate-sensitive aggregates appear to be an extremely difficult family to optimize in our context.

Result from PIER

Figure 5.7 depicts the total network communication savings achieved by our duplicate insensitive techniques executing over the PIER as a function of the total gain achievable in the query workload. In this specific experiment, PIER was configured to use the Bamboo DHT over $1,024$ nodes, and run standard hierarchical MAX queries; furthermore, each node had a randomly-generated data distribution such that all nodes share the same F matrix and explicit synchronization was not needed.

The numbers shown are for a workload of 100 concurrent MAX aggregation queries. The "individual queries" line shows the baseline communication overhead when the multi-query optimization feature is not utilized. The "non-optimized" line uses the multi-query optimization feature, but *does not perform any actual sharing*, and instead uses the identity fragment matrix for F' — the line just serves to illustrate the communication overhead of multi-query optimization (essentially, the overhead of shipping the longer fragment identifiers). Finally, the last line employs our Add-MAP optimization strategy, demonstrating a very substantial, linear decrease in communication cost as the achievable gain in the workload increases.

F Size	Algorithm	A' size	% Effective	Runtime (ms)
50x100	Basic-M/A/W	60.9 (17.3)	53.6%	33304 (16694)
50x100	Add-O/A	57.6 (16.3)	58.4%	28416 (20980)
50x100	Add-O/R	48.6 (12.2)	73.1%	19622 (4780)
50x100	Add-O/S	49.4 (11.8)	72.0%	9059 (3484)
50x100	Add-M/A/D	55.6 (16.4)	61.5%	35145 (13880)
50x100	Add-M/A/P	45.2 (11.8)	78.3%	2730 (733)
50x100	Add-M/A/W	52.6 (14.3)	66.5%	5190 (2035)
50x100	Add-M/R/D	87.6 (5.2)	18.4%	33409 (16544)
50x100	Add-M/S/D	49.7 (12.7)	71.4%	12675 (4488)
50x100	Add-M/R/P	87.9 (5.1)	18.1%	33475 (16116)
50x100	Add-M/S/P	87.9 (5.1)	18.1%	33145 (15982)
50x100	Add-M/R/W	87.6 (5.2)	18.4%	33424 (16580)
50x100	Add-M/S/W	99.8 (0.4)	0.3%	22610 (1681)
100x100	Basic-M/A/W	77.8 (16.5)	47.5%	38740 (14815)
100x100	Add-O/A	77.1 (20.3)	41.4%	36493 (21889)
100x100	Add-O/R	67.3 (20.7)	72.2%	26433 (15700)
100x100	Add-O/S	66.7 (20.3)	74.1%	9924 (6624)
100x100	Add-M/A/D	77.3 (20.5)	41.1%	45602 (23751)
100x100	Add-M/A/P	64.6 (21.2)	79.6%	4170 (1737)
100x100	Add-M/A/W	71.7 (20.4)	58.0%	6975 (2265)
100x100	Add-M/R/D	96.5 (4.7)	6.0%	56812 (28320)
100x100	Add-M/S/D	67.7 (20.0	72.3%	14727 (12835)
100x100	Add-M/R/P	96.3 (5.1)	6.2%	56416 (27898)
100x100	Add-M/S/P	96.3 (5.1)	6.2%	56453 (27972)
100x100	Add-M/R/W	96.5 (4.7)	6.0%	56754 (28308)
100x100	Add-M/S/W	99.6 (0.8)	1.5%	18984 (9136)
150x100	Basic-M/A/W	77.3 (15.5)	48.6%	44446 (18225)
150x100	Add-O/A	78.0 (19.8)	40.8%	46702 (28646)
150x100	Add-O/R	67.6 (20.4)	71.1%	39248 (25552)
150x100	Add-O/S	65.8 (20.5)	76.1%	11672 (8922)
150x100	Add-M/A/D	77.9 (20.1)	40.6%	59397 (36990)
150x100	Add-M/A/P	64.6 (21.1)	79.9%	5788 (2554)
150x100	Add-M/A/W	71.9 (20.1)	57.6%	8694 (3107)
150x100	Add-M/R/D	96.7 (4.4)	6.0%	67981 (31839)
150x100	Add-M/S/D	67.7 (20.2)	73.2%	18458 (16314)
150x100	Add-M/R/P	96.5 (5.0)	6.3%	68117 (32045)
150x100	Add-M/S/P	96.5 (5.0)	6.3%	68173 (32209)
150x100	Add-M/R/W	96.7 (4.4)	6.0%	67929 (31739)
150x100	Add-M/S/W	99.6 (1.0)	1.5%	22105 (12201)

Table 5.2: Data for all duplicate sensitive algorithms averaged over all total gains for each particular matrix size. Standard deviation shown in parentheses.

Chapter 6

Related Work

PIER is the first and 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.

6.1 Internet Systems

The leading example of a massively distributed system is the Internet itself. The soft-state consistency of the Internet's internal data [14] is one of the chief models for our work. On the schema standardization front, we note that significant effort is expended in standardizing protocols (e.g. IP, TCP, SMTP, HTTP) to ensure that the "schema" of messages is globally agreed-upon, but that these standards are often driven by popularly deployed software. While rarely stored persistently, the number of bytes generated from each of these "schemas" annually is enormous.

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., [42]). These systems have proved effective for their narrow workloads, though there are persistent concerns about DNS on a number of fronts [58].

As is well known, P2P filesharing is a huge phenomenon, and systems like KaZaA, Gnutella, eDonkey, and BitTorrent each have hundreds of thousands of users at any given time. 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., [49, 84]).

6.2 Database Systems

Query processing in traditional distributed databases focused on developing bandwidth-reduction schemes, including semi-joins and Bloom joins, and incorporated these techniques into traditional frameworks for query optimization [61]. Mariposa was perhaps the most ambitious attempt at geographic scaling in query processing, attempting to scale to thousands of sites [75]. Mariposa focused on overcoming cross-administrative barriers by employing economic feedback mechanisms in the cost estimation of a query optimizer. To our knowledge, Mariposa was never deployed or simulated on more than a dozen machines, and offered no new techniques for query *execution*, only for query optimization and storage replication. By contrast, we have deferred work on query optimization, preferring to first design and validate the scalability of our query execution infrastructure.

Many of our techniques here are adaptations of query execution strategies used in parallel database systems [21]. Unlike distributed databases, parallel databases have had significant technical and commercial impact. While parallelism per se is not an explicit motivation of our work, algorithms for parallel query processing form one natural starting point for systems processing queries on multiple machines.

PIER's architecture and algorithms are closer to parallel database systems like the Gamma System [20], Volcano [26], 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. Some algorithms, such as Bloom joins originated from the IBM research project R* [53], which was the distributed version of System R.

In terms of its data semantics, PIER most closely resembles centralized data integration and web-query systems like Tukwila [40] and Telegraph [70]. 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 area of distributed stream query processing, an application that PIER supports. The Borealis proposal [13] 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 [5] augments this vision with Mariposa-like economic negotiation among a few large agents.

The HiFi project [22] discusses the challenges with high fan-in systems. In particular they examine RFID applications which generate large amounts of data at the edge of the network. Through a hierarchy of processing the quantity of data is reduced to an amount suitable for a single centralized system. The project focuses on removing anomalies, interpolating missing data, removing duplicates, and validating the stream of readings.

Tian and DeWitt presented analytical models and simulations for distributed eddies [76]. 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. [7] 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.

Somewhat more tangential are proposals for query processing in wireless sensor networks [10, 54]. These systems share our focus on peer-to-peer architectures and minimizing network costs, but typically focus on different issues of power management, extremely low bandwidths, and very lossy communication channels.

6.3 Distributed Hash Tables

Most traditional massively scalable query systems (e.g. DNS, P2P filesharing) use a hierarchical scheme to enhance recall. Hierarchical architectures raise concerns about scalability, availability, and resilience to attack, since they display centralized characteristics towards the root of the hierarchy.

By contrast, DHTs [63, 74, 87, 68] provide a flat, truly peer-to-peer topology for distributed lookups. Different DHT schemes differ in their details, but all provide common facilities: the ability to partition a value domain across multiple machines, to efficiently (in few network hops) route messages by value to the node responsible for the value's partition, to provide some transient storage and retrieval of these messages, and to efficiently reconstruct the topology in the face of frequent node arrivals and departures.

6.4 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 [30]. Another early vision of P2P databases was presented by Bernstein et al. [8], 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 was a main focus of the Piazza project; a representative result is their work on mediating schemas transitively as queries propagate across multiple databases [34]. 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 [39]. These research agendas are complementary with PIER's. An early effort in this regard is the PeerDB project [59], 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 [24] 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 [78] 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 shared these goals was Sophia [80], a distributed Prolog system for network information. Sophia's vision was essentially a superset of PIER's, inasmuch as the relational calculus is a subset of Prolog. Sophia never developed distributed optimization or execution strategies, the idea being that such functionality could be coded in Prolog as part of the query. Another workshop proposal for peer-to-peer network monitoring software is presented in [72], including a simple query architecture, and some ideas on trust and verification of measurement reports.

P2 [48] is another distributed dataflow system. P2 uses a novel high-level declarative language, NDLog [47], which is based on Datalog. The initial work of the P2 project is focused on implementing overlay networks (including DHTs) and other network protocols using P2. NDLog provides for recursive queries (which are required for most network protocols) and automatic execution optimization. This enables the implementation of complex overlay networks in 100x fewer lines of code. The initial development of NDLog used PIER before the P2 team developed their own dataflow system.

Range indexing is another topic that is being explored in multiple projects (e.g., [18, 9, 33], 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 [18], it works over any DHT (unlike Mercury [9]), 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.

6.5 Multiquery Optimization

Much of the prior work on multi-query optimization (such as [69]) focuses on select/project/join queries. Our work addresses aggregation, which was not widely addressed in prior research.

For the case of a *single* distributed aggregation query, efficient in-network execution strategies have been proposed by several recent papers and research prototypes (including, for instance, TAG [54], SDIMS [83]). The key idea in these techniques is to perform the aggregate computation over a dynamic tree in an overlay network. Aggregation occurs over a dynamic tree, with each node combining the data found locally along with any *Partial State Records (PSRs)* it receives from its children, and forwarding the resulting PSR one hop up the tree. Over time, the tree dynamically adjusts to changing node membership and network conditions. More recent work on distributed data streaming has demonstrated that, with appropriate PSR definitions and combination techniques, in-network aggregation ideas can be extended to fairly complex aggregates, such as approximate quantiles [17, 29], and approximate histograms and join aggregates [16]. None of this earlier work considers the case of multiple distributed aggregation queries, essentially assuming that such queries are processed individually, modulo perhaps some simple routing optimizations. For example, PIER suggests using distinct routing trees for each query in the system, in order to balance the network load [38].

In the presence of hundreds or thousands of continuous aggregation queries, system performance and scalability depend upon effective sharing of execution costs across queries. Recent work has suggested

solutions for the *centralized* version of the problem, where the goal is to minimize the amount of computation involved when tracking (1) several group-by aggregates (differing in their grouping attributes) [86], or (2) several epoched aggregates (differing in their epoch sizes and/or selection predicates) [44, 46], over a continuous data stream *observed at a single site*. In the distributed setting, network communication is the typical bottleneck, and hence minimizing the network traffic becomes an important optimization concern.

In an independent effort, [77] proposes a distributed solution for linear aggregates. Their scheme is based on heuristics tailored to power-constrained sensornets where the query workload is restricted to a *static* collection of simple spatial predicates related to the network topology. Instead, our dynamic fragment-based method does not have any restrictions on the query predicates, and employs optimal linear-algebra techniques to uncover sharing across linear aggregates. They also observe the analogy to the Set-Basis problem for `MIN/MAX` aggregates but do not propose any algorithmic solution for the duplicate-insensitive case.

Chapter 7

Conclusion

In this thesis we presented a query processor designed to aggressively use DHTs to implement many traditional DBMS functions. 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. Similarly, the DHT can also be used to create secondary hash indexes (as used in the symmetric semi-join in Section 4.3.2).
- **Range Index Substrate:** The PHT technique provides resilient distributed range-search functionality by mapping the nodes of a trie search structure onto a DHT [65].
- **Partitioned Parallelism:** Similar to the Exchange operator [26], 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 need to maintain their own separate hash tables.
- **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 in Section 4.2 and joins in Section 4.3).

We showed that PIER can be used to efficiently execute aggregation and join queries. Aggregation queries can be executed using a wide-range of strategies. We discussed five dimensions to classify the vari-

ous strategies, including the structure of the aggregation communication, whether in-network aggregation is performed, the routing method, the dynamism of the routing, and the timing of communication.

Our results showed that using the DHT's routing alone to derive the aggregation tree is not efficient. By using the DHTs routing and additional query processing logic we were able to construct a more efficient tree that eliminates network bottlenecks. Furthermore, by exposing the topology to the application we were able to choose timing that reduces latency and extra network communication.

We showed that for executing joins the use of the DHT imposes some modest overhead, the DHT provided a feasible and scalable platform for executing joins. We presented a number of algorithms (symmetric hash join and Fetch Matches) and rewrite strategies (symmetric semi-join and Bloom joins). We showed their performance tradeoffs of the different strategies under different workloads.

Finally, in Chapter 5 we showed that in an environment where there are many simultaneous aggregation queries, we can further optimize the execution to significantly reduce network communication. We developed novel algorithms to optimize hundreds of simultaneous queries and explored the performance and effectiveness of them.

Overall we presented a complete system that achieves the goal of providing a rich query language with location transparency and scalability with relaxed semantics. The design, architecture, and algorithms presented in thesis can be used to build Internet-scale, data intensive applications.

Appendix A

UFL Language

UFL is a low level syntax for writing queries for execution in PIER. UFL is loosely based on *nix object-oriented configuration files.

The grammar for UFL is listed below:

- `<Query> ::= (<Object>)+`
- `<Object> ::= '%' <ObjectType> '(' <ObjectName> ')' '{' (<Object> | <Setting>)* '}'`
- `<ObjectType> ::= 'opgraph' | 'operator' | 'predicate'`
- `<ObjectName> ::= <TextString>`
- `<Setting> ::= <Key> ('[' (<Index>)? ']')? '=' <Value>`
- `<Key> ::= <TextString>`
- `<Value> ::= <LiteralTextString> | <Expression>`
- `<Index> ::= <TextString>`
- `<Expression> ::= <ExpressionConstant> | <ExpressionField> | <ExpressionFunction>`
- `<ExpressionConstant> ::= <LiteralTextString> '::' <CastType>`
- `<CastType> ::= <TextString>`
- `<ExpressionField> ::= '$' <LiteralTextString>`
- `<ExpressionFunction> ::= <FunctionName> '(' (<Parameter> (',' <Parameter>)*)? ')'`

- `<FunctionName> ::= <LiteralTextString>`
- `<Parameter> ::= <Expression>`
- `<TextString> ::= (['a'-'z', 'A'-'Z', '0-9', '.', '*', '-', '_', '/'])+`
- `<LiteralTextString> ::= <TextString> | '"' (<LiteralChar> | <LiteralEscape>)* '"'`
- `<LiteralChar> ::=` Any char except \or '
- `<LiteralEscape> ::= '(<Unicode> ('$')? | <QueryExpansion> ('$')? | '\$' | '"' ('$')?)`
- `<Unicode> ::= 'u' ['0-9', 'A-F', 'a-f'] ['0-9', 'A-F', 'a-f'] ['0-9', 'A-F', 'a-f'] ['0-9', 'A-F', 'a-f']`
- `<QueryExpansion> ::= 'qt' | 'qi'`

There are three types of objects: opgraph, operator and predicate. Any operator object must be embedded within an opgraph object. Likewise any predicate object must be embedded in an operator object. Although the grammar does not specifically enforce the nesting, the UFL parser does. All objects are named which can then be used as references to that object.

Settings are key-value pairs that belong to the object they are specified in. Settings can have flat, array or hash table values. Each element of the array or hash table is listed as a separate setting with a different index. To illustrate the types of settings:

- **Flat:** No index is specified

  ```
  type = SCAN
  ```

- **Array:** Index represents position in array, starting with zero. The elements may be listed in any order. Elements with no index specified will be appended to the end of the array in the order listed.

  ```
  RenameList[0] = Column1
  RenameList[40] = Column2
  RenameList[2] = Column3
  RenameList[] = Column4
  ```

- **Hash Table:** Index represents the key for the entry in the hash table.

  ```
  Options[FE] = 100
  Options[FW] = 20
  Options[FC] = 32
  ```

The type (hash table or array) of the setting is inferred based on the first use. If the first listing of the setting uses an integer typed index value it is assumed to be an array, otherwise it is a hash table. To force

a hash table structure even when using numeric keys, a dummy entry having an index of '*' and any value is specified.

The parser recognizes a number of settings for each type of object. A partial list for each object is listed below:

- **opgraph:**
  - `ns` the namespace for the query dissemination, optional.
  - `rid` the resource identifier for the query dissemination, optional.
  - `duration` a long integer representing the duration of execution in seconds.
- **operator:**
  - `type` the type of operator: cache, dupelim, eddy, flowcontrol, groupby, join, null, projection, put, queue, result, scan, selection, tee or union.
  - `implementation` the specific implementation of the operator. This permits multiple implementations of the same operator.
  - `sources` - an array representing the parent operators
  - There are also operator type specific settings.
- **predicate:**
  - `type` the type of predicate or set, valid values include: and, or, or atomic
  - `leftexpression` - an expression (as defined by the grammar), only valid if the type is atomic.
  - `rightexpression` - an expression (as defined by the grammar), only valid if the type is atomic
  - `op` the predicate operator, valid values include: =, !=, ¿, ¿=, ¡, ¡=, equals, notequals, greaterthan, greaterthanequals, lessthan, and lessthanequals, only valid if the type is atomic.

Appendix B

One-Shot Aggregation Algorithms

- **IP-1L-Optimal.** The query specifies the root using an IP socket address and the number of nodes, n, participating in the query. Every node directly send a single PSR containing the aggregate of all local data to the root using a direct IP message. After the root receives and aggregates n values the query is complete and the result is forwarded to the requester. Note that n is generally not known in large dynamic system.

- **IP-1L-Timeout.** The query specifies the root using a IP socket address and a timeout condition. As with **IP-1L-Optimal** every node sends a single PSR to the root using a direct IP message. Once the timeout condition is satisfied at the root the result is produced. If additional data arrives after the timeout, the timeout condition is reset and a revised result is produced when the next timeout occurs.

- **DHT-1L-Optimal.** The query specifies the root using a DHT ID and the number of nodes, n, participating in the query. Every node sends a single PSR to the root's DHT ID using the DHT put function. The root uses the $lScan$ and $newData$ functions to receive the PSRs. After the root receives n values the result is produced.

- **DHT-1L-Timeout.** The query specifies a root using a DHT ID and a timeout condition. As with **DHT-1L-Optimal** every nodes send a single PSR to the root's DHT ID using the DHT put function. Once the timeout condition is satisfied at the root, the result is produced. If data arrives after the timeout, the timeout condition is reset and a revised result is produced when the next timeout occurs.

- **DHT-ML-Optimal.** The query specifies a root using a DHT ID and the number of nodes, n, participating in the query. Every node sends a single PSR towards the root using the DHT $send$ function. Each node also listens for PSRs using the DHT $upcall$. When a node other than the root receives a PSR, the node simply forwards the PSR to the next hop using $send$ without any additional processing. After the root receives n values the result is produced. This algorithm will cause interior nodes to send multiple PSRs.

- **DHT-ML-Timeout.** The query specifies a root using a DHT ID and a timeout condition. As with **DHT-ML-Optimal** every node sends a single PSR towards the root's DHT ID using the DHT $send$ function and forward any PSRs they receive. Once the timeout condition is satisfied at the root, the result is produced. If additional data arrives at the root after the timeout, the timeout condition is reset and a revised result is produced when the next timeout occurs. This algorithm will cause interior nodes to send multiple PSRs.

- **DHT-Hi-Optimal.** The query specifies a root using a DHT ID. In addition, the query must specify the number of nodes that each interior node should receive data from (not just the root). This requirement is just looser than specifying the topology in the query since interior nodes just need to be informed how many nodes they should receive PSRs from, not the IDs/IPs of those nodes. Every leaf node sends a single PSR towards the root's DHT ID using the DHT $send$ function. Each node listens for PSRs using the DHT $upcall$ method. When a node receives the specified number of PSRs, it combines the PSRs using the aggregation function and forwards a single PSR to the next hop using the $send$. When the root receives the specified number of PSRs the result is produced. This algorithm is not supported by PIER since the topology generated by the DHT is not known *a priori* so the expected number of messages can not be pre-determined.

- **DHT-Hi-Timeout.** The query specifies a root using a DHT ID and a timeout condition. As with **DHT-Hi-Optimal** every node sends a single PSR towards the root's DHT ID using the DHT $send$ function, and listens for PSRs using the DHT $upcall$ method. Once the timeout condition is reached at a node, the node combines the PSRs using the aggregation function and forwards a single PSR to the next hop using the $send$. When the timeout condition is reached at the root the result is produced. This algorithm will cause interior nodes will send at least two PSRs with nodes higher in the tree sending more (number of messages is equal to the node's height in the tree where leaf nodes have height one). If at a node (including the root) data arrives after the timeout, the timeout condition is reset and a revised PSR (or result) is produced.

Appendix C

Continuous Aggregation Algorithms

- **IP-1L-Optimal**, **IP-1L-Timeout**, **DHT-1L-Optimal**, and **DHT-1L-Timeout.** Same algorithms as for one-shot queries except the same process is repeated for every epoch with each message containing an epoch field.

- **IP-1L-Learned.** The query specifies a root using an IP socket address and a child cleanup timeout value. For each epoch every node directly sends a single PSR containing the aggregate of all local data to the root. The root maintains a list of children and after receiving PSRs from all children (all other nodes) produces the answer for that epoch. Children are removed from the list after not sending any PSRs within the specified child cleanup timeout.

- **DHT-1L-Learned.** The query specifies a root using a DHT ID and a child cleanup timeout value. For each epoch every node directly sends to the root DHT ID using the DHT put function. The root uses $lScan$ and $newData$ to receive PSRs. The root maintains a list of children and after receiving PSRs from all children (all other nodes) produces the answer for that epoch. Children are removed from the list after not sending any PSRs within the specified child cleanup timeout.

- **DHT-ML-Optimal** and **DHT-ML-Timeout.** Same algorithms as for one-shot queries except the same process is repeated for every epoch with each message containing an epoch field.

- **DHT-ML-Learned.** The query specifies a root using a DHT ID and a node cleanup timeout value. For each epoch every node sends a single PSR towards the root's DHT ID using the DHT $send$ function. Each node also listens for PSRs using the DHT $upcall$. When a node other than the root receives a PSR it immediately forwards the PSR to the next hop using $send$. The root maintains a list of nodes and after receiving PSRs from all nodes produces the answer for that epoch. Nodes are removed from the list after not sending any PSRs within the specified node cleanup timeout. This algorithm will cause interior nodes to send multiple PSRs.

- **DHT-ML-S-Optimal.** The query specifies a root using a DHT ID and the number of nodes, n, partic-

ipating in the query. On the first epoch every node sends a single PSR towards the root using the DHT $send$ function remembering the ID of the next hop node. On subsequent epochs the node will send its PSR to the same node using the stored ID from the first epoch. If that node is no longer reachable, the DHT will automatically route the message to the node with the closest ID. Each node also listens for PSRs using the DHT $upcall$. When a node other than the root receives a PSR, is immediately forwards the PSR to the next hop using $send$. After the root receives n values the answer is produced. This algorithm will cause interior nodes to send multiple PSRs. Since the DHT API does not report the actual next hop ID[1], this algorithm is not implemented in PIER.

- **DHT-ML-S-Timeout.** The query specifies a root using a DHT ID and a timeout condition. As with **DHT-ML-S-Optimal**, in the first epoch the PSR is routed to the root ID using the DHT $send$ function remembering the ID of the next hop node, subsequent epochs send to that same ID unless the node is not reachable, and each node forwards any PSRs it receives. When the timeout condition is reached at the root the result is produced. If data arrives at the root after the timeout, the timeout condition is reset and a revised answer is produced when the next timeout occurs. This algorithm requires that interior nodes send multiple PSRs. Again, since the DHT API does not report the actual next hop this algorithm is not implemented in PIER.

- **DHT-ML-S-Learned.** The query specifies a root using a DHT ID and a node cleanup timeout value. As with **DHT-ML-S-Optimal**, in the first epoch the PSR is routed to the root's DHT ID using the DHT $send$ function remembering the ID of the next hop node, subsequent epochs send to that same ID, and all non-root nodes immediately forward any PSRs they receive. The root maintains a list of nodes and after receiving PSRs from all nodes, the root produces the result for that epoch. Nodes are removed from the list after not sending any PSRs within the specified node cleanup timeout. This algorithm will cause interior nodes to send multiple PSRs. Again, since the DHT API does not report the actual next hop this algorithm is not implemented in PIER.

- **Tree-ML-Optimal.** The query specifies a root using a DHT ID, the number of nodes of nodes, n, participating in the query, and the desired maximum number of children (or fan-in of a node). Similar to **DHT-ML-Optimal**, in the first epoch every node sends a single PSR towards the root using the DHT $send$ function. Each node also listens for PSRs using the DHT $upcall$. When a node other than the root receives a PSR, it simply forwards the PSR to the next hop using $send$. After the root receives n values the answer is produced. If at any time during or at the end of the epoch a node detects that the node has more than the maximum number of children, it will send a message to the excess children requesting that they use contact a different parent in the next epoch. The ID of a non-excess child is included in the message. If a node receives this redirect message from its parent, it will send subsequent PSRs to the new ID instead of to the root's ID. On failure of the new parent, the node will resume sending to the

[1] The DHT is unable to return the next hop ID synchronously since the next hop is not known until the message is sent, received by the remote node, and ACKed. The DHT could optimistically return the first hop choice synchronously, but this may be wrong if that node is now unreachable and an alternate route is used.

root's ID. This algorithm requires that interior nodes send multiple PSRs, and that parents may send messages to their children.

- **Tree-ML-Timeout.** The query specifies a root using a DHT ID, a timeout condition, and the desired maximum number of children. As with **Tree-ML-Optimal** in the first epoch the PSR is routed to the root ID using the DHT $send$ function., nodes listen for PSRs using the DHT $upcall$, and non-root nodes simply forward the PSRs to the next hop using $send$. If at any time a node detects that it has more than the maximum number of children, it sends a message to its excess children requesting they use the ID of a non-excess child as their parent in subsequent epochs until the new parent fails. When the timeout condition is reached at the root the result is produced. If data arrives after the timeout, the timeout condition is reset and a revised answer is produced when the next timeout occurs. This algorithm requires that interior nodes send multiple PSRs and that parents may send messages to their children.

- **Tree-ML-Learned.** The query specifies a root using a DHT ID, a node cleanup timeout value, and the desired maximum number of children. As with **Tree-ML-Optimal** in the first epoch the PSR is routed to the root ID using the DHT $send$ function, nodes listen for PSRs using the DHT $upcall$, and non-root nodes simply forward the PSRs to the next hop using $send$. If at any time a node detects that it has more than the maximum number of children, it sends a message to its excess children requesting they use the ID of a non-excess child as their parent in subsequent epochs until the new parent fails. The root maintains a list of nodes and after receiving PSRs from all nodes, the root produces the result for that epoch. Nodes are removed from the list after not sending any PSRs for the specified node cleanup timeout. This algorithm requires that interior nodes send multiple PSRs and that parents may send messages to their children.

- **Tree-ML-S-Optimal.** The query specifies a root using a DHT ID, the number of nodes, n, participating in the query, and the desired maximum number of children (or fan-in of a node). On the first epoch every node sends a single PSR towards the root using the DHT $send$ function remembering the ID of the next hop node. On subsequent epochs the node will send its PSR to the same node using the stored ID from the first epoch. If that node is no longer reachable, the DHT will automatically route the message to the node with the closest ID. Each node also listens for PSRs using the DHT $upcall$. When a node other than the root receives a PSR, is immediately forwards the PSR to the next hop using $send$. If at any time a node detects that it has more than the maximum number of children, it sends a message to its excess children requesting they use the ID of a non-excess child as their parent in subsequent epochs until the new parent fails. After the root receives n values the answer is produced. This algorithm will cause interior nodes to send multiple PSRs.

- **Tree-ML-S-Timeout.** The query specifies a root using a DHT ID, a timeout condition, and the desired maximum number of children (or fan-in of a node). As with **Tree-ML-S-Optimal**, in the first epoch the PSR is routed to the root ID using the DHT $send$ function remembering the ID of the next hop

node, subsequent epochs send to that same ID unless the node is not reachable, and each node forwards any PSRs it receives. If at any time a node detects that it has more than the maximum number of children, it sends a message to its excess children requesting they use the ID of a non-excess child as their parent in subsequent epochs until the new parent fails. When the timeout condition is reached at the root the result is produced. If data arrives at the root after the timeout, the timeout condition is reset and a revised answer is produced when the next timeout occurs. This algorithm requires that interior nodes send multiple PSRs.

- **Tree-ML-S-Learned.** The query specifies a root using a DHT ID, a node cleanup timeout value, and the desired maximum number of children (or fan-in of a node). As with **Tree-ML-S-Optimal**, in the first epoch the PSR is routed to the root's DHT ID using the DHT $send$ function remembering the ID of the next hop node, subsequent epochs send to that same ID, and all non-root nodes immediately forward any PSRs they receive. If at any time a node detects that it has more than the maximum number of children, it sends a message to its excess children requesting they use the ID of a non-excess child as their parent in subsequent epochs until the new parent fails. The root maintains a list of nodes and after receiving PSRs from all nodes, the root produces the result for that epoch. Nodes are removed from the list after not sending any PSRs within the specified node cleanup timeout. This algorithm will cause interior nodes to send multiple PSRs.

- **DHT-Hi-Optimal** and **DHT-Hi-Timeout.** Same algorithms as for one-shot queries except the same process is repeated for every epoch with each message containing an epoch field.

- **DHT-Hi-Learned.** The query specifies a root using a DHT ID and a child cleanup timeout value. As with **DHT-Hi-Optimal** every node sends a single PSR towards the root's DHT ID using the DHT $send$ function, and listens for PSRs using the DHT $upcall$ method. Each node maintains a list of children and after receiving PSRs from all children the node combines the PSRs using the aggregation function and forwards a single PSR to the next hop using $send$. If a PSR is not received within the child cleanup timeout value the child is removed from the list and the PSR is sent to the next hop. If a PSR from a child not on the list is received, or a second PSR is received from any existing child after the PSR for that epoch has already been sent, an additional PSR is sent.

Bibliography

[1] Tbit, the tcp behavior inference tool. `http://www.icir.org/tbit/`.

[2] 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.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[4] 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.

[5] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Load Management and High Availability in the Medusa Distributed Stream Processing System. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Paris, June 2004.

[6] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, Jack Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadalphia: Society for Industrial and Applied Mathematics. Also available as postscript file on http://www.netlib.orgtemplatesTemplates.html, 1994.

[7] 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.

[8] 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.

[9] 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.

[10] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *Proc. of the IEEE International Conference on Mobile Data Management (MDM)*, volume 1987 of *Lecture Notes in Computer Science*, Hong Kong, January 2001. Springer.

[11] 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.

[12] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. Telegraphcq: Continuous dataflow processing for an uncertain world, 2003.

[13] 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.

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

[15] Saar Cohen and Yossi Matias. Spectral bloom filters. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 241–252, New York, NY, USA, 2003. ACM Press.

[16] Graham Cormode and Minos Garofalakis. "Sketching Streams Through the Net: Distributed Approximate Query Tracking". In *Proc. of the International Conference on Very Large Data Bases (VLDB)*, Trondheim, Norway, September 2005.

[17] Graham Cormode, Minos Garofalakis, S. Muthukrishnan, and Rajeev Rastogi. "Holistic Aggregates in a Networked World: Distributed Tracking of Approximate Quantiles". In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, June 2005.

[18] 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.

[19] 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.

[20] 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.

[21] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 35(6):85–98, 1992.

[22] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, Fredrick Reiss, Shariq Rizvi, Eugene Wu, Owen Cooper, Anil Edakkunni, and Wei Hong. Considerations for high fan-in systems: The hifi approach. In *CIDR*, Jan 2005.

[23] Sumit Ganguly, Minos Garofalakis, and Rajeev Rastogi. Processing set expressions over continuous update streams. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 265–276, New York, NY, USA, 2003. ACM Press.

[24] 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.

[25] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM SIGACT News*, 33(2), June 2002.

[26] 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.

[27] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge. Discovery*, 1(1):29–53, 1997.

[28] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[29] Michael B. Greenwald and Sanjeev Khanna. "Power-Conserving Computation of Order-Statistics over Sensor Networks". In *Proc. of the ACM Symposium on Principles of Database Systems (PODS)*, Paris, France, June 2004.

[30] 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.

[31] Krishna Gummadi, Ramakrishna Gummadi, Steve Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of dht routing geometry on resilience and proximity. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 381–394, New York, NY, USA, 2003. ACM.

[32] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop (IMW 2002)*, Marseille, France, November 2002.

[33] 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.

[34] 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.

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

[36] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proc. of the 1997 ACM SIGMOD international conference on Management of data*, pages 171–182. ACM Press, 1997.

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

[38] Ryan Huebsch, Brent N. 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. In *CIDR*, Jan 2005.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, June 2006.

[45] 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.

[46] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, 2005.

[47] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David A. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking: Language, execution and optimization. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Chicago, June 2006.

[48] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. October 2005.

[49] 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.

[50] 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.

[51] Anna Lubiw. The boolean basis problem and how to cover some polygons by rectangles. *SIAM Journal on Discrete Mathematics*, 3(1):98–115, 1990.

[52] Carsten Lund and Nihalis Yannakakis. "On the Hardness of Approximating Minimization Problems". *Journal of the ACM*, 41(5), September 1994.

[53] 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.

[54] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, December 2002.

[55] 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.

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

[57] Napster. `http://en.wikipedia.org/wiki/Napster`.

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

[59] 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.

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

[61] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999.

[62] 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.

[63] 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.

[64] 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.

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

[66] 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.

[67] Martin Roesch. Snort – Lightweight Intrusion Detection for Networks. In *Proc of the 13th USENIX Systems Administration Conference (LISA)*, Seattle, WA, November 1999.

[68] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218, 2001.

[69] Timos K. Sellis. Multiple-query optimization. In *Proc. of the ACM Transactions of Database Systems*, volume 13, pages 23–52, 1988.

[70] 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.

[71] 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.

[72] Sridhar Srinivasan and Ellen Zegura. Network Measurement as a Cooperative Enterprise. In *Proc. of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[73] L. J. Stockmeyer. The minimal set basis problem in NP-complete. Technical Report RC 5431, IBM Research, May 1975.

[74] 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.

[75] 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.

[76] 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.

[77] Niki Trigoni, Yong Yao, Alan J. Demers, Johannes Gehrke, and Rajmohan Rajaraman. Multi-query optimization for sensor networks. In *Distributed Computing in Sensor Systems (DCOSS)*, pages 307–321, Marina del Rey, CA, 2005. Springer.

[78] 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.

[79] 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.

[80] 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.

[81] 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.

[82] 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.

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

[84] 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.

[85] 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.

[86] Rui Zhang, Nick Koudas, Beng Chin Ooi, and Divesh Srivastava. Multiple aggregations over data streams. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 299–310, New York, NY, USA, 2005. ACM Press.

[87] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.