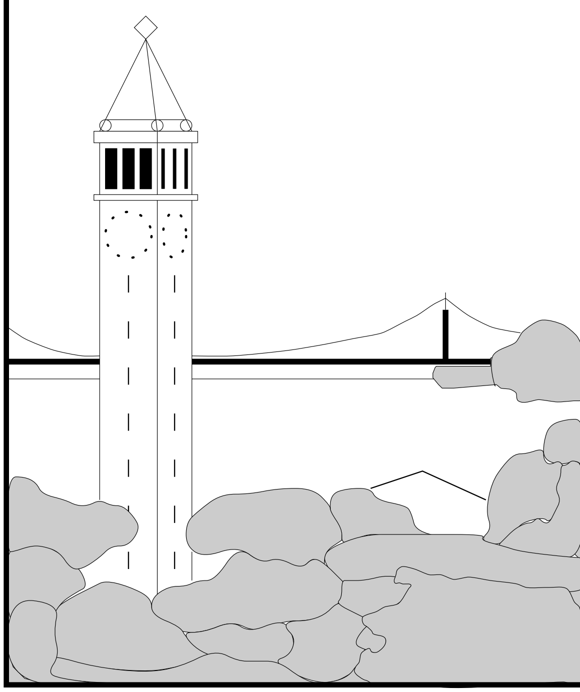
FREddies: DHT-Based Adaptive Query Processing via FedeRated Eddies

Ryan Huebsch and Shawn R. Jeffery
EECS Computer Science Division, UC Berkeley
{huebsch, jeffery}@cs.berkeley.edu



Report No. UCB/CSD-4-1339

July 2004

Computer Science Division (EECS)
University of California
Berkeley, California 94720

FREddies: DHT-Based Adaptive Query Processing via FedeRated Eddies

Ryan Huebsch and Shawn R. Jeffery
EECS Computer Science Division, UC Berkeley
{huebsch, jeffery}@cs.berkeley.edu

July 2004

Abstract

In response to the ever increasing scale, distribution, and complexity of data processing, database research in the past few years has focused on adaptive query processing. However, many of these solutions, although aimed at processing wide-area data, remain centralized solutions. In this paper, we present FREddies, an extension of the centralized Eddy operator for use in a P2P query processing system. FREddies operate within the framework of PIER, a DHT-based P2P query processor. FREddies optimize the query during runtime and require no global knowledge. We show that FREddies using rudimentary routing policies can perform competitively with a traditional static query optimization approach. Furthermore, we validate our simulation results in the real world environment of PlanetLab.

1 Introduction

As society progresses further into the information age, the corpus of online information is quickly expanding, seemingly without bounds. Current technology is only able to provide simple (keyword) query facilities. Distributed query systems are beginning to appear; however their usefulness is limited by their ability to execute complex queries efficiently in the wide-area.

The PIER project [7], a peer-to-peer (P2P) relational query processor, is designed for complex querying at a global scale. However, due to the complex nature of the data and the queries, query optimization methods from previous peer-to-peer systems are not useful. Furthermore, the fully decentralized nature of PIER limits the applicability of traditional database optimization techniques [11]. Finally, data sources in the wide area exhibits widely changing characteristics: sources may be slow, have bursts, or be unresponsive.

The emergence of adaptive query processing, most notably of which is the Eddy [2], shows great promise for applications in a P2P environment. An Eddy is able to dynamically adjust the flow of tuples through the query plan, thus reacting at runtime to fluctuations in data arrival rate, data distributions, and available resources. Until now, the Eddy operator has been implemented only in a centralized database system [4].

We have developed the FREddy (FedeRated Eddy) to perform query optimization in PIER. A FREddy is a query plan operator that dynamically routes tuples through local operators (some of which may send the tuple to another node in the network).

Through its routing choices, the FREddy is able to decide a query plan for each tuple on-the-fly.

Although the FREddy may not be as efficient as some centralized solutions, we explore its benefits when implemented from a P2P purist's perspective. Developing equivalent centralized solutions are likely to encounter numerous engineering challenges, such as maintaining a global dynamic catalog. True P2P systems subvert this engineering challenge by sacrificing the need for global knowledge (and efficiency in some cases).

As an initial implementation and study, we focus on showing the feasibility of FREddies for adaptive query processing in a P2P query processor. Our contributions are:

- Develop and implement the FREddy mechanism within the PIER system
- Evaluate the FREddy's base performance and overhead through a comparison with possible static plans
- Show the potential for complex routing policies by demonstrating the benefit of a simple routing policy's performance above the baseline

The remainder of the paper is structured as follows. Section 2 details other research efforts in distributed query processing. Background on both the PIER system and the Eddy operator is given in Section 3. Section 4 presents the FREddy operator and its functionality within PIER. Section 5 briefly introduces a variety of routing policies. We present our experimental results in Section 6 and conclude in Section 7.

2 Related Work

Related research can be divided into two categories: distributed optimization and centralized adaptive query processing.

SwAP (Scalable and Adaptable query Processor) [16] is also based on the Eddy. To replicate the lottery scheme introduced in the original Eddy paper [2], they introduce the concepts of *remote operator* and *virtual tuple*. The virtual tuple provides explicit feedback to the Eddy about selectivities of remote operators. FREddies are similar to SwAP, however we are able to harness the horizontal partitioning of the DHT. SwAP has an initial preparatory phases which is centralized, FREddies have no equivalent process.

In [13], the authors explore various routing policies for distributed Eddies. Their focus is on routing policies between distributed operators, not a full implementation of Eddies. Their effect on performance in a static setting. Their results will be useful in developing routing polices for FREddies. In addition their work focuses on vertical parallelism, while FREddies are able to take advantage of both horizontal and vertical parallelism.

Adaptive query processing for centralized systems has been addressed by work including [4, 14, 8]. The Telegraph project uses the Eddy to perform query optimization. The project is focused on multi-query optimization over heterogeneous streams. FREddies is the natural extension of their work.

Query scrambling is a technique for changing the query plan during processing. Periodically during processing, the query can be re-optimized. Execution is temporarily stopped and the plan is changed (some cleanup work must also be performed). Each query optimization still utilizes traditional centralized techniques; however, catalog information may be updated between each optimization.

Finally, the Tukwila project uses adaptive techniques for processing XML streams in a centralized system. Their work utilizes incremental re-optimization for multi-query optimization.

3 Background

In this section we introduce the reader to PIER and the Eddy.

3.1 PIER

PIER is a fully decentralized query processor designed to scale from thousands to millions of nodes. PIER sacrifices some traditional database features such as storage and transactions and focuses solely on query processing.

Designed for P2P environments, PIER is a best-effort system. Ensuring consistency (such as ACID) is prohibitively expensive in the wide-area. In addition, P2P applications are unlikely to depend on precise answers. Here we briefly describe PIER's foundations. For further reading on PIER and its functionality, please see [7].

One of the key design features of PIER is the use of a distributed hash table (DHT) as the communication substrate. DHTs form an overlay routing layer where data can be directly addressed instead of just nodes. Objects (messages) are assigned a key, usually by hashing its name or the contents. The object (message) is then stored (routed) to the node currently responsible for that key. The mapping of keys to nodes is dynamically maintained in the presence of nodes entering and leaving the system.

To achieve scalability, each node maintains information about a small subset other nodes. Routing is then achieved via a DHT specific algorithm that requires multiple hops [1, 9, 12, 10, 15]. In most cases, the number of hops is bounded by the logarithm of the number nodes in the system.

PIER is a generic data flow engine; however, it was designed for executing relational queries. A key advantage of relational systems is the ability to specify queries in a declarative language such as SQL. Like a traditional database, queries must be parsed and optimized to create a physical query plan which can be executed. The physical query plan specifies exactly which operators are used and the order and interconnection between them. Relation queries can be executed in a number of *logically* equivalent ways. For the correct processing of some operators, a tuple may be moved from one node to another node [1].

Previous to this work, PIER had no form of optimization and required the user to submit a pre-optimized query plan.

3.2 Eddy

A substantial portion of a database system is dedicated to the optimization stage. Poorly optimized queries can execute orders of magnitudes slower than good plans. Optimizers utilize the system catalogs which contain statistics about tables and columns to assist in choosing the query plan.

For example, consider the following query:

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.b AND
       S.c = T.d;
```

There are two possible join orders, compute the join of R and S first followed by T, or compute S join T followed by R (a diagram of these plans can be found in Figure 3). Although both plans are logically equivalent, the best ordering is generally the one that produces the fewest intermediate results. As the number of tables in the query are increased, the number of possible join orders grows exponentially.

The Eddy eliminates the need to choose a static ordering. An Eddy operator is placed between all of the operators. Figure 1 illustrates a simple query plan

[1] One such case is processing joins, where the tuple may have to be sent to the node responsible for a particular join value. For example, a join between R and S on $R.a = S.b$, all R tuples with value x in field a must be sent to the same site as all S tuples with value x in field b (recall the join condition is $R.a = S.b$).

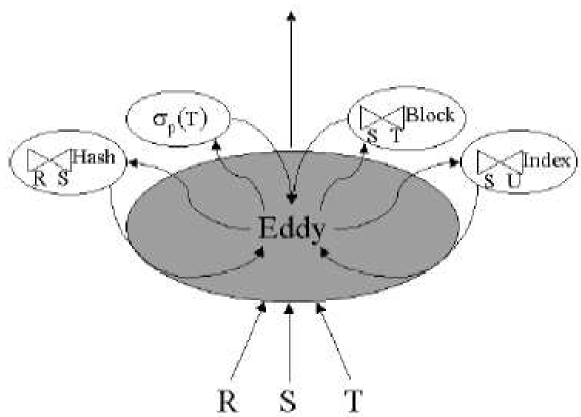


Figure 1: A centralized Eddy performing a four table join and one selection. One of the tables, U, is accessed via an index. Image is from [2].

with an Eddy. When a tuple arrives at the Eddy, the Eddy consults a *routing policy* to determine the next operator that should receive the tuple.

In order to enable tuples to be routed individually, each tuple must have some additional state with which it is associated. At a minimum this state includes a bitmap of $DoneBits$. Each bit represents an operator in the plan. The bits are initially unset (0). When a tuple is processed by an operator the appropriate bit is set (1). When all of the bits are set, the tuple can be routed to the output.

By itself, the Eddy will not insure correctness. Each of the joins must be carefully executed to insure all results are produced. The interested reader is invited to read [2]. For the purposes of this paper, PIER's use of hash-based ripple join algorithms suffices to insure correctness.

4 FREddies

Standard optimizers require access to a catalog of statistics. These statistics reflect size, distribution, and location of data. As mentioned earlier, one of the primary tenants of PIER is its purist P2P nature. A centralized catalog would not satisfy that requirement. Therefore, any optimizer used with PIER should also be decentralized.

It is conceivable to construct a distributed catalog, however this poses a number of additional challenges. Maintaining consistency in a distributed catalog can become expensive in the face of many updates. Such a catalog would also have to be continuously accessible, otherwise new queries could not be optimized. Finally, the entire catalog must be viewable to insure the optimizer can choose a good plan. If only partial information is available the optimizer is more likely to make an invalid assumption and produce a poor plan. Based on [3], such a distributed catalog is not possible. The catalog must be consistent, available, and tolerate partitioning. The CAP principle states that only two of the three properties are achievable at any given time. Even if a distributed catalog were feasible, it still does not solve the optimization problem completely.

An important challenge with designing wide-area systems is tolerance for dynamic conditions. During query execution, network links go down or become congested or real-time data streams may vary in data makeup. Thus, assumptions made at the beginning of the query may not hold through the lifetime of the query.

A P2P environment also implies that nodes are heterogeneous in processing capabilities, which may change from one moment to the next. Furthermore, nodes are constantly entering and leaving the network (churn). Thus the available *compute* resources are constantly changing. A query engine must adapt by changing the locations where queries are processed.

In addition to changing data and processing conditions, there is no guarantee that those conditions are uniform across all nodes. It is likely that local conditions in one part of the network are different from

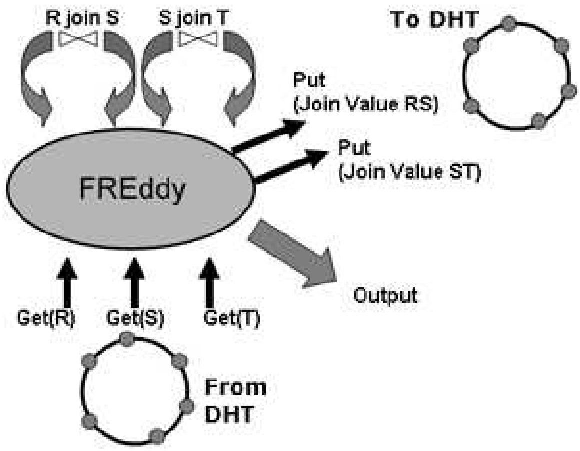


Figure 2: An example of FREddy query plan performing a three table join. For simplicity some operators have been removed. In this example, the FREddy may route to local operators (joins) or rehash tuples to other nodes (`put`) for processing. In this case, the FREddy's main decision is to which `put` (RS or ST) to send a tuple.

the global conditions. A query plan that is optimal on some nodes may be unacceptable for other nodes.

For all of these reasons, an adaptive query processor is the best solution for a distributed query system. The Eddy is a natural solution since it was designed for heterogeneous/federated data sources. Based on the Eddy, we have developed the FREddy, a distributed version of the Eddy.

4.1 Query Processing with the FREddy

In essence, a FREddy is a local routing operator which moves tuples between operators. An operator may accept tuples for processing and/or return (or create) tuples for additional processing. The FREddy is placed in the middle of the query plan such that it intercepts any tuples an operator has outputted, and sends it to another operator. A unique instance of the FREddy is executed on every node processing the query. Each FREddy will make independent routing decisions. See Figure 2 for an example FREddy query.

The first phase in FREddy-based query processing is plan creation. The origin node determines which operators are necessary to execute a query as well as computes the possible operator routing orders. In essence, this step creates a join spanning tree that avoids cross-products, which are logically valid but almost always inefficient. Using our ongoing 3-way join example, an R tuple should not be processed by the ST-join until it is processed by the RS-join. However an S tuple may be processed by either the RS-join or the ST-join in either order. The acceptable order information is encoded in the query plan.The same query plan is used by all nodes participating in the query. In this way, all nodes share execution information and processing is simplified.

The query is then disseminated in the same manner as any other query in PIER. Once a participating node receives a query, it instantiates all operators including the FREddy. Inter-operator data pipes are created connecting each operator to the FREddy. The FREddy then begins dataflow by requesting tuples from each of its sources.

In order to support per-tuple routing, each tuple in a FREddy-based query plan is tagged with metadata indicating the operators it has been processed by. The FREddy operator (and not each subordinate operator) does the metadata management, thus allowing a FREddy to be used with any existing operator without modification.

This metadata, referred to as the $DoneBits$, is a bit array of length N, where N is the number of operators that accept tuples in the query. For the average query, the metadata will be on the order of one or two bytes. When a new tuple arrives into the Eddy (one with no metadata) the FREddy tags the tuple's metadata with an empty $DoneBits$ array.

When a tuple arrives at the FREddy, the $DoneBits$ and the routing policy are consulted to determine the next operator. We discuss the routing policy in more detail in Section 5. Some operators may be network I/O operators, which move tuples between FREddies (nodes). In this way, network messages are abstracted from the FREddy, such as in [6]. Since each FREddy was initialized the same, this is a seamless process.

When all $DoneBits$ of a tuple are set, the tuple is routed to the output operator which returns the tuple to the origin.

5 Routing Policies

The routing policy is used to determine the next operator for a tuple. At a minimum, the routing policy is able to use each tuple's metadata to make its decision. However, the policy may also keep statistics and even communicate with other nodes to improve the quality of its decisions.

In many cases the routing policy will be optimized to be efficient for some metric, such as throughput, response time, or resource consumption. In PIER, the bottleneck resource is almost always network communication. We expect that many policies will be optimized to reduce network communication which should benefit both response time and system throughput.

One straightforward heuristic is for the routing policy to insure that a tuple is processed through all applicable local operators before sending the tuple to an operator that may send it over the network [2].

The design space for potential routing policies is large. We only discuss two simple policies and introduce one slightly more complicated routing policies to show the potential for smarter policies. More complicated policies are beyond the scope of this initial paper.

[2]Although this may seem to be *always* correct, it is possible that a local operator may create more work (i.e. tuples) for future processing. Thus it is possible that always routing to local operators is short sighted.

5.1 Static Policy

The simplest routing policy is one where the FREddy chooses the next operator based on the order specified in the query. The primary usages for this policy are debugging the FREddy code and determining the overhead of the FREddy mechanism.

5.2 Random Policy

A random routing policy is the simplest dynamic policy. The policy will choose the next possible local operator at random. Only after all applicable local operators have processed the tuple will the policy send the tuple to an operator that will send the tuple over the network.

The effect of this policy is that roughly equal numbers of tuples will be routed through every possible join ordering. Therefore, this plan is expected to have average performance; it will be faster than the worse plans but slower than the best plans. It is important to note that in a P2P environment without accurate, global knowledge, this is an acceptable option, although obviously not optimal.

5.3 Queue Length Policy

As an illustrating example of both the potential and complexities of routing policies, we introduce the Queue Length (QL) routing policy. By observing information available locally, this policy attempts to learn global conditions.

Suppose each operator in the system had an ingress queue. For operators such as selection, projection, and join, the queue length would represent the number of tuples the FREddy has routed to that operator, but the operator has not processed yet. For operators such as `put` which involve network operations, the queue length represents the number of tuples currently waiting to be sent, in-flight, or are waiting for the DHT at the remote node.

A routing policy that monitors the queue length of local `put` operators will be able to provide better load balancing and overall performance by estimation of a variety of query and data characteristics.

A growing queue for a `put` operator is an indication that either the intervening network is congested or the remote node has not been able to keep up with the influx of tuples. Therefore, queue length for the `put` operator is an indirect means of measuring the load on the network and remote node. For example, suppose the RS-join is expensive, and produces many more tuples than its input, while the ST-join is highly selective and its output is small. Nodes that are processing the RS-join are likely to be more heavily loaded and doing more network communication, i.e. moving the joined tuples to other nodes after processing, while the nodes running the ST-join will be doing less work and communication. Therefore, nodes processing the ST-join should have shorter queue lengths and it is desirable to route tuples to it first.

Furthermore, monitoring the queue length for `puts` allows the FREddy to estimate the global distribution of the join key. Because FREddies run over a DHT, a tuple is hashed on its join key, which routes the tuple to the node responsible for that join bucket. This has the effect of sending all tuples of equal value to a single node. Thus, if the cardinality of distinct join keys for a particular relation is small, many tuples will be rehashed to a small number of nodes. This will cause these nodes' queues to grow if they cannot handle the large number of incoming tuples. For load balancing purposes, tuples should be routed away from these nodes.

Additionally, such a join key distribution is usually an indication that a join that will produce a large number of results. Ideally, tuples should be sent elsewhere in hopes they will be discarded before being sent to such a join.

Our QL routing policy is based on these premises. We assume it is always best to process to local operators first before sending the tuple over the network. Thus the policy will only maintain queue lengths for the `put` operators. A counter is stored for each `put` operator. The counter is incremented with every tuple routed to the operator, and decremented when the DHT returns the acknowledgment that the `put` has been completed successfully.

When choosing between multiple `put` operators, the FREddy will route the tuple to the operator with the shortest queue length. We show in Section 6 that this capable of performing noticeably better than the random policy.

6 Experimental Results

We implemented the FREddy operator and the above routing policies in Java as part of the PIER query operator framework. Overall, the FREddy code was approximately 500 lines of non-commented source statements, while PIER has about 5000 lines total.

To determine the viability of our approach, we measured FREddy performance in PIER. The goals of our experiments are three-fold:

1. Determine the overhead that the FREddy mechanism imposes
2. Compare the FREddy performance to that of a static query optimizer
3. Determine how FREddies perform in a real-world environment

For the bulk of our experiments, we used the use the PIER network simulator. The simulator is able to model message-level network delays and clock timers. CPU processing costs are not modeled; however, we expect network costs to dominate processing costs in our system. The network topology was a simple star-topology where congestion is modeled for the inbound and outbound traffic on each node. It is assumed the middle of the network has been overprovisioned. The latency between nodes was set at 100ms.

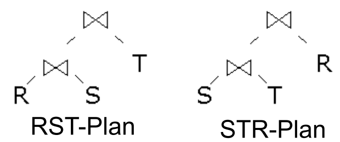


Figure 3: Enumerations of the possible join orderings for a 3 table join query.

For our simulations, we ran 256 PIER nodes running over Chord. Chord was selected for the DHT layer mainly because it is faster to simulate. Although our simulator could handle more nodes and more complex topologies, the goal of our experiments was to show the feasibility of FREddies, not their scalability.

We ran two different queries, one three table join and one four table join:

```
SELECT *
  FROM R, S, T
 WHERE R.a = S.b AND
       S.c = T.d;
```

and

```
SELECT *
  FROM R, S, T, U
 WHERE R.a = S.b AND
       S.c = T.d AND
       T.e = U.f;
```

There are two possible join orderings for the first query and five for the second query. We enumerate the plans in Figure 3 and 4.

For the 3-way join query, the RS-join was designed to create many output tuples, having a selectivity of 25 (each input tuple will match 25 other tuples) and 10% of the R and S tuples will not find any matches. The ST-join was designed to highly selective, having a selectivity of 2 and 60% of the R and S tuples were discarded. Thus, an oracle would choose to execute the ST join first to eliminate as many tuples as possible before sending the output to the RS join.

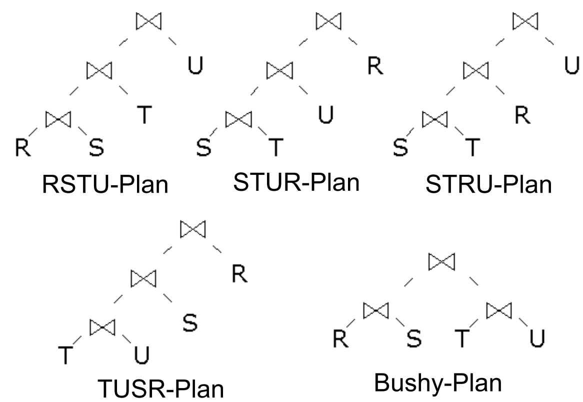


Figure 4: Enumerations of the possible join orderings for a 4 table join query

For the 4-way join, the ST join was the expensive join, producing 10 tuples for each input. The RS and TU joins were cheaper, producing 1 tuple for 40% of its inputs and 2 tuples for 60% of its inputs, respectively. The optimal plan is to execute the RS join first, followed by the ST and finally TU (cross products are avoided). Each table is preloaded with 256,000 tuples (100 tuples per node).

6.1 Base Performance

Our first experiment was to measure the overhead of using the FREddy. We ran the 3-table join with both the RST and the FREddy with the static routing policy configured to execute the same as the RST plan.

Our simulation is only able to quantify the cost of the additional metadata that must travel with the tuple over the network. Although the FREddy may require more CPU cycles our simulator does not cap-

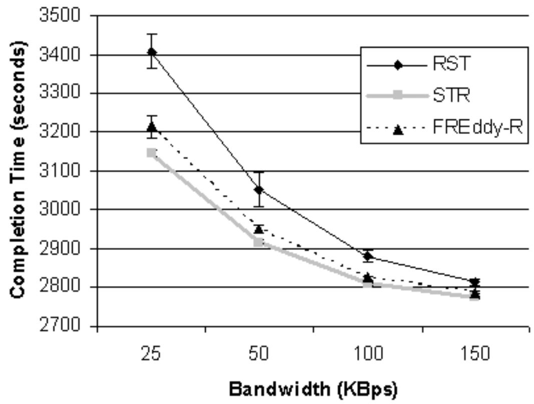


Figure 5: Completion time for the 3-way join at various node bandwidths

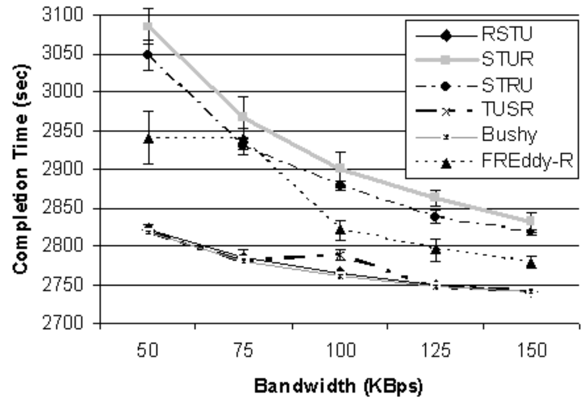


Figure 6: Completion time for the 4-way join

ture those costs[3]. Furthermore, it has been shown in [5] that, correctly implemented, the Eddy mechanism imposes negligible processing overhead.

Our simulation shows that the RST uses 159MB of aggregate network traffic, while the FREddy uses 186MB. There is approximately 70 bytes of overhead for each tuple due to the metadata. Note that this is an unoptimized implementation and it is expected that this number can be decreased.

Our next two experiments show that the FREddy with a random routing policy performs relatively well for both a three table join (Figure 5) and a four table join (Figure 6). We varied the bandwidth (at each node) to show how the long it takes to complete the query with varying degrees of network congestion.

In both cases, the FREddy is in the middle (as expected), performing better than the worst plans and slightly worse than best plans.

[3]Recall that the expected bottleneck is network congestion, so CPU is not an important cost to measure, assuming it is reasonable.

Plan	**Avg. #puts**	**Avg. B.W.**	**Time**	**Std. Dev.**
RST	280K	104MB	2935	17.11
STR	154K	47MB	2899	1.85
FREddy-R	217K	84MB	2909	2.56
FREddy-QL	163K	71MB	2902	2.28

Table 1: Continuous publishing simulation results, comparing the static plans with the random FREddy and a FREddy using the QL routing policy

6.2 Queue Length Routing Policy

Our next experiment was designed to show that the FREddy is capable of performing even better when a smarter routing policy is used. For this experiment the tables are not preloaded, and are continuously published in small batches beginning shortly after the query is executed and continuing until all 256,000 tuples are inserted.

Not only is this a more realistic query in our target environment, but it gives the routing policy opportunity to learn query characteristics. If all the data is preloaded, then the FREddy will be forced to make a decision on every tuple all at once, preventing it from

Plan	Min.	Max.	Avg.	Std. Dev.
RST	42.54	164.67	111.52	56.77
STR	36.20	70.36	52.03	13.07
FREddy-R	49.42	214.98	103.80	65.98
FREddy-QL	58.54	59.53	59.04	0.69

Table 2: Time to the 5000th tuple on PlanetLab. Plans RST, STR, and FREddy-R were run five times, while FREddy-QL was only run twice

seeing the consequences of any of its decisions.

Table 1 shows the results. We show the total number of DHT `puts` executed, the total aggregate network traffic during the query, the time till query completion, and finally the standard deviation for the completion time.

The data shows that the FREddy-QL (QL routing policy) performs almost as well as the best query plan (RST). The number of `puts` is slightly higher, showing that the FREddy-QL does make some wrong decisions. There is a larger difference in aggregate bandwidth due to the extra metadata the FREddy-QL must send.

These results just scrape the surface of more advanced routing policies. However, it is interesting to see that a very basic policy, using information that is obtained locally, is able to come very close to the performance of the best static plan.

6.3 PlanetLab Results

Our last set of experiments were performed on PlanetLab. For these experiments, approximately 180 nodes were used. We use Bamboo as the DHT layer because its implementation handles network failures better. The same data and queries as in the simulations were run. Table 2 shows the results.

The table shows data for the arrival of the 5000th tuple. Although the queries produce more results due to conditions out of our control, not all queries returned all the results. To compensate, we look at the 5000th result, which represents about half of the overall results.

As can be seen from the standard deviations reported, the data was not very consistent between runs. This is be expected with the PlanetLab testbed. However, the results do indicate the FREddy is performing relatively similar to our simulations.

7 Conclusion

In this paper we explore the need for adaptive query processing within a P2P system and propose a variant of the Eddy, a FREddy, to perform tuple by tuple routing decisions. Our initial experiments show the feasibility of our design. Finally, we show the promise of more complicated routing policies by showing the performance of one simple routing policy (Queue Length).

It is expected that this work will be continued and will become the standard method of optimizing queries in PIER.

References

[1] Bamboo. *Submitted for publication*, 2003.

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

[3] E. Brewer. Invariant boundaries. In *Ninth International Workshop on High Performance Transaction Systems (HPTS)*, Oct. 2001. Presentation available at `http://research.microsoft.com/~jamesrh/hpts2001/`.

[4] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman,

F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[5] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 2003.

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

[7] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *Proc. of VLDB 2003*, Sept. 2003.

[8] Z. G. Ives, A. Y. Levy, D. S. Weld, D. Florescu, and M. Friedman. Adaptive query processing for internet applications. In *IEEE Data Engineering Bulletin*, volume 23, June 2000.

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

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

[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, Boston, May 1979.

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

[13] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of VLDB 2003*, pages 333–344, Sept. 2003.

[14] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proc. 1998 ACM SIGMOD International Conference on Management of Data*, pages 130–141, 1998.

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

[16] Y. Zhou. Adaptive distributed query processing. In *PhD Workshop VLDB 2003*, Sept. 2003.