An Initial Study of Overheads of Eddies

Amol Deshpande

Computer Science Division
University of California, Berkeley, CA
amol@cs.berkeley.edu

Abstract

An eddy [2] is a highly adaptive query processing operator that continuously reoptimizes a query in response to changing runtime conditions. It does this by treating query processing as *routing* of tuples through *operators* and making per-tuple routing decisions. The benefits of such adaptivity can be significant, especially in highly dynamic environments such as data streams, sensor query processing, web querying, *etc*. Various parties have asserted that the cost of making per-tuple routing decisions is prohibitive. We have implemented eddies in the PostgreSQL open source database system [1] in the context of the TelegraphCQ project. In this paper, we present an "apples-to-apples" comparison of PostgreSQL query processing overhead with and without eddies. Our results show that with some minor tuning, the overhead of the eddy mechanism is negligible.

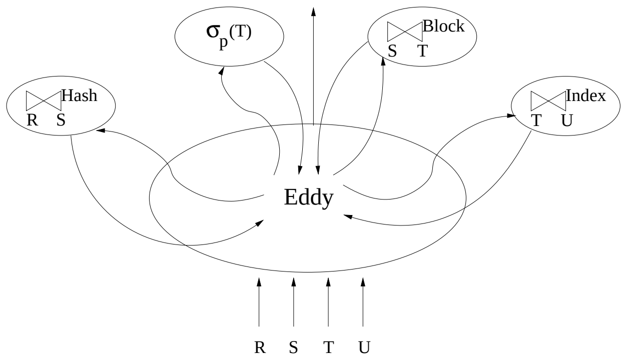

Figure 1: [2] An eddy in a pipeline. Data flows into the eddy from input relations R, S, T and U. The eddy routes tuples to operators, that in turn return tuples to the eddy. The eddy sends a tuple to the output only when it has been handled by all the operators. The eddy adaptively chooses an order to route each tuple through the operators.

1 Introduction

Adaptive query processing has emerged as an attractive solution in many scenarios with dynamically changing runtime environments and unknown data distributions. These scenarios often involve remote networked data sources, including sensors, data streams, web sources. Adaptive query processing has also been proposed for single site database systems where often only unreliable statistics are available. There has been a good deal of work on this topic, much of it surveyed in [5].

Eddies [2, 11, 10, 12, 3] represent the most adaptive technique proposed for such environments. The basic idea behind eddies is to model query execution as *routing* of tuples among *operators* and to adapt to dynamically changing runtime conditions by making independent routing decisions per tuple (Figure 1). Eddies offer tremendous flexibility in adapting to dynamic runtime conditions at a very fine granularity. However, various parties have raised performance concerns about the overhead of making per-tuple routing decisions, raising questions about the effectiveness of eddies.

In this paper, we describe an implementation of the eddies architecture in the PostgreSQL 7.3 open-source database system, and present an experimental study that demonstrates how such an adaptive architecture can be implemented with minimal overheads. This work was done in the context of the TelegraphCQ project [7], which is building a highly adaptive, sharing-oriented query processor for continuous queries. TelegraphCQ is designed to process multiple queries simultaneously while exploiting sharing of work. In this paper, we focus on single query execution (also the focus of the original work on eddies [2]). A more complete discussion of the TelegraphCQ query processor is given in [7, 6].

1.1 Rest of the Paper

We begin with describing how we implemented this architecture in the PostgreSQL data management system, and how we minimize the overhead of making routing decisions (Section 2). In Section 3, we present an initial experimental study that demonstrates the viability of the eddies architecture in a real system. We conclude with a discussion of work in progress in the TelegraphCQ project (Section 4).

2 Implementation Details

In this section, we describe the changes that we made to the PostgreSQL database system to implement eddy support for a *single* query at a time. This is in contrast to TelegraphCQ, which focuses on multi-query eddy sharing. In order to do an apples-to-apples comparison of overheads, we wanted to focus strictly on the overhead of eddies as compared to traditional query processing schemes. Hence we ignore novel multi-query optimizations and continuous query scenarios that are not supported by PostgreSQL. However we stress that the eddy architecture we describe here also directly supports (single-query) stream processing.

2.1 New Operators

We added four new operators to PostgreSQL, an *eddy* operator, a *selection* operator, a *symmetric hash join* operator and an *eddy index join* operator. The eddy uses the latter three operators to execute a query, by routing tuples through them. We begin with describing the implementation of these operators (that can also be used in traditional PostgreSQL plans) first, and then move on to describing the eddy operator. Many of the implementation details are direct consequences of our design decision to follow the PostgreSQL query processing infrastructure as closely as possible. In particular, we decided to keep the system single-threaded, and we implemented the new operators to support the traditional iterator interface [8] that is also used by PostgreSQL. This was done in order to leverage the existing PostgreSQL operators as much as possible, and to be able to use eddies in more complex query plans.

2.1.1 The *Selection* Operator

PostgreSQL does not have an explicit selection operator for evaluating selection predicates. Instead, each operator in PostgreSQL has a list of attached predicates that are evaluated in order. The PostgreSQL optimizer attaches the predicates as low in the query plan as it can, with the lists on each operator ordered arbitrarily. By adding explicit selection operators, we can allow an eddy to consider reordering multiple, possibly expensive selections, and even interleave them after joins.

2.1.2 The *Symmetric Hash Join* Operator

Eddies do best with operators that have frequent "moments of symmetry" [2]. We added a symmetric hash join operator to PostgreSQL to exploit this; this join operator can be used in traditional PostgreSQL query plans as well.

The symmetric hash join operator maintains two hash tables on the two relations involved in the join, and supports the traditional *get_next()* interface by encapsulating all the state inside the operator itself. When an operator upstream performs a *get_next()* operation on this operator, it takes following steps :

- If there is an outstanding *probe* tuple for which all matches have not yet been returned, return the *next* match for that probe tuple. To find this *next* matching tuple, the operator remembers the last matching tuple that was returned for this probe tuple.
- If there are no outstanding tuples, then the operator "obtains" a new input tuple, *inserts* it into the appropriate hash table, and *probes* into the other hash table. Depending on the context in which this operator is being used, this new tuple may be "obtained" in two different ways :
  - If the operator is being used in a static (traditional) PostgreSQL plan, then it will obtain the next input by calling *get_next()* on either of its two input operators.
  - If eddies are being used, then the eddy operator will provide (route) the next input to the operator (as we will see in Section 2.1.4). The operator returns *null* in this case to signify that it needs a new input tuple.

2.1.3 The *Eddy Index Join* Operator

The eddy index join operator is quite similar to the traditional index join operator in PostgreSQL and exports a similar *get_next()* interface, except that :

- It uses a different tuple format (Section 2.2).
- The next *outer* tuple (also called a *probe* tuple), is provided by the eddy operator; in traditional PostgreSQL, the index join operator will call *get_next()* on its child operator to obtain the next probe tuple.

2.1.4 The *Eddy* Operator

Each eddy operator handles a simple select-project-join statement, consisting of arbitrary joins, selections and projections, but no aggregations or group-bys or subqueries. These latter operations are executed by existing PostgreSQL operators, that can be used without any modifications as the eddy operator supports the iterator interface used by these operators. Figure 2 shows an example of how an eddy is instantiated for a query involving aggregation. Treating eddy as just another operator also allows simultaneous use of multiple eddy operators in a single query execution plan, with each eddy operator handling a part of the query that consists of only select, project and join operators.

Creating a Plan with Eddys

The PostgreSQL plan creation routine starts by identifying *query blocks* that consist of only selects, projects, and

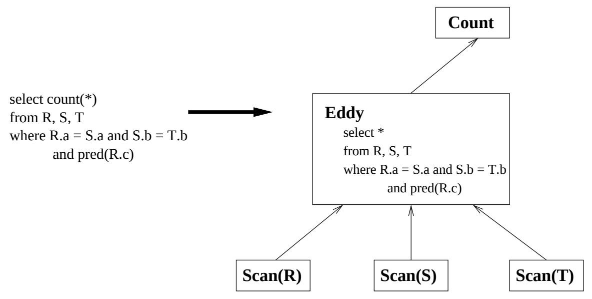

Figure 2: Using traditional operators along with an eddy

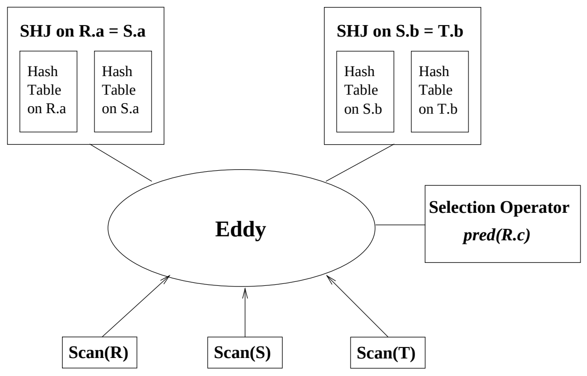

Figure 3: Eddy instantiated for the example query

joins. It then calls the PostgreSQL cost-based optimizer to choose and construct an execution plan for each such block. It also instantiates operators for handling the rest of the SQL constructs such as aggregates and group-bys, that are then put together with the execution plans for the query blocks to obtain an execution plan for the entire query. To be able to use eddies, we modified the plan creation routine so that, instead of calling the optimizer, it creates an *eddy* operator directly for each query block (as discussed above, such a query block is exactly what a single eddy operator can handle as well). These eddy operators are then put together with operators instantiated for handling aggregations and group-bys as before, to get an execution plan for the entire query.

In addition, to set up each select-project-join query block with an eddy, the plan creation routine instantiates a set of operators as follows :

- For each data source in the query block (these may be base relations, streams or subqueries), a *Scan* operator is instantiated. The eddy interacts with these operators using the iterator interface, and as such, the exact nature of these operators is not relevant.
- For each expensive predicate in the query, the parser instantiates a selection operator. Predicates without expensive methods are attached to operators as in traditional PostgreSQL.
- For each join in the query, either a symmetric hash join operator or an index join operator is instantiated. For our experiments here, we let the PostgreSQL optimizer choose the join algorithm, and ensure that the algorithm choices are the same when comparing static plans to eddies.

Figure 3 shows the operators instantiated by the query for the example query.

Routing Arrays

The eddy executes a query by fetching tuples from the source operators, and routing the tuples through the selection and join operators. The eddy maintains a set of *routing arrays* that help it in deciding how to route the tuples. The routing arrays essentially maintain all the possible operators that a tuple of a given *signature* may be routed to, where the signature of a tuple is defined to be the sources it consists of and the operators it has already passed through. As suggested in [2], the signature of a tuple is stored in the form of a bitmap along with the tuple itself. We use these bitmaps to index the routing arrays in order to quickly find all the operators a tuple of a given signature can be routed to. The number of such signatures (or bitmaps) is usually exponential, and as such, this approach is not suitable for queries with a large number of relations. We are currently exploring alternative approaches for such queries.

Query Processing

To support the iterator model interface, the eddy needs to remember the computational state it was in when it returned the *last* tuple. The eddy does this by maintaining a *stack* of *active* operators. The join operators are called *active* if they have an outstanding probe tuple; as we will discuss below, selection operators are treated differently during query execution.

When an upstream operator performs a *get_next()* operation on the eddy, the eddy performs the following steps :

1. If there are any active operators (if the stack is not empty), call *get_next()* on the operator on the top of the stack.
   - If the call returns a *null*, pop the operator off the stack, and repeat Step 1.
   - If the *get_next()* call returns a tuple, proceed to Step 3.
2. If there are no active operators, choose a source from the list of input sources and use *get_next()* to fetch a new source tuple. If none of the sources have any more data, finish processing by returning *null*.
3. When a new tuple is returned to the eddy (from either a source or an operator), check if it is an output tuple (signified by an empty routing array entry). If it is an output tuple, return it to the caller. Otherwise, (a) use the routing array in conjunction with the routing policy (discussed below) to choose an operator to route this tuple to, (b) route the tuple to

the operator, and (c) push the operator on the top of the stack (selection operators are applied directly to the tuple to avoid unnecessary function calls). Repeat step 1.

Routing Policies

All the machinery that we have discussed up to this point is quite general, and accommodates a wide variety of routing *policies*. A routing policy takes a tuple, and chooses a particular operator as the next routing destination for the tuple. The routing arrays maintained by the eddy provide the policy with a quick way to find all the next semantically valid operators for a tuple.

The lottery scheduling scheme proposed in the original work on eddies [2] can be implemented by maintaining ticket information with each operator, using the routing array to find all the operators a tuple can go to, and randomly choosing among those operators via appropriate weights. Note that the *backpressure* technique proposed in that work cannot be directly used in our system because of the single-threaded nature of our system. Instead, we explicitly instrument the processing costs of operators at runtime, allowing us to capture the relative processing rates without backpressure.

Even though the cost of a scheme like lottery scheduling is not very high, we can reduce these overheads even further, by amortizing the costs across a batch of tuples. This is done by making routing decisions for all tuple signatures (for all possible bitmaps) only periodically, and adhering to those decisions for a period of time. As an example, if the query contains a set of selections on a single relation, instead of choosing the order of applying the selections for each tuple, we choose a single order and use it for a certain number of tuples.

The results of routing policy decisions are maintained in the routing arrays themselves, by keeping the next operator of choice for a signature at the beginning of the signature's list of operators. Making the routing decision for a given tuple, in this case, simply involves choosing the first operator in the routing array entry for the bitmap of the tuple.

By batching in this manner, we can trade off the cost of adaptivity against the time-granularity of adaptivity. This not only allows us to hide the overhead of routing policies like lottery scheduling, it also allows us to consider using more sophisticated reoptimization techniques, whose costs would have been prohibitive if they are invoked for every tuple. We will call the periodicity at which these decisions are made the *reoptimization frequency*.

Current Routing Policy

The routing policy that we use in this paper is based on *rank* ordering [9]. For each tuple signature, and for each operator it can be routed to, we estimate the *selectivity*, *i.e.*, the number of tuples that will be generated if the tuple is routed to that operator. The eddy maintains statistics in order to compute these estimates. These selectivities are then used to update the routing array by ordering the operators for each tuple signature using them. The number of tuple signatures is exponential in the number of sources to be joined, and as such, this process can be fairly expensive for some queries.

2.2 Tuple Format

The other major change to the PostgreSQL query engine that we make pertains to the way tuples are stored, and operated upon, during execution. Every tuple in PostgreSQL has an associated *tuple descriptor* that references the catalog information for attributes present in the tuple. In the traditional PostgreSQL implementation, the tuple descriptors for the inputs to an operator are all statically determined at optimization time, and this fact is used to dereference the attributes of tuples during the operator's execution. With eddies, the tuple descriptors handled by an operator are likely to change during execution, since the signatures of input tuples are not fixed. For example, some of the input tuples to an operator may have already been joined with tuples from other tables, while others may have not. As a result, the default tuple format of PostgreSQL is not usable with eddies.

Rather than change all the code in PostgreSQL to use a different format, we chose a hybrid approach. The operators attached to an eddy use an indirection mechanism on top of the PostgreSQL tuple format. The eddy and its attached operators represent each tuple as an array of pointers to actual source tuples (that are in default PostgreSQL format). The code that dereferences fields in tuples is changed so that it knows how to use this indirection. This format is only used inside the operators attached to the eddy; the tuples are converted into this format when they are brought inside the eddy, and are converted back to default PostgreSQL format when they are sent outside the eddy.

3 Experimental Results

In this section, we will present an initial experimental study that validates our claims that the overhead of the eddy architecture is not very high. Our focus is on measuring only the overhead of eddies, and as such we use artificial queries for which *every* execution plan has *identical* cost. As a result, these experiments do not penalize the traditional PostgreSQL system for its inability to adapt. This provides a worst-case analysis of the overhead of our eddy implementation, and separates our measurements of overheads from any issues controlled by the effectiveness of policies.

3.1 Setup

All the experiments were run on a lightly loaded 2GHz Pentium 4 machine running Redhat Linux 8.3. We use two sets of artificial tables for our experiments, *tenk1-5* and *hundredk1-5* with 10,000 and 100,000 244-byte

No-Delay
```
SELECT count(*)
FROM tenk1
WHERE two < 2
and four < 4
and ten < 10
```

Delay-X, X = 0, 10, 100
```
SELECT count(*)
FROM tenk1
WHERE delay(two, X) < 2
and delay(four, X) < 4
and delay(ten, X) < 10
```

Selection Expt Queries

Select-All-3
```
SELECT *
FROM tenk1, tenk2, tenk3
WHERE tenk1.unique2 = tenk2.unique2
and tenk1.unique2 = tenk3.unique2
```

Select-Count-3
```
SELECT count(*)
FROM tenk1, tenk2, tenk3
WHERE tenk1.unique2 = tenk2.unique2
and tenk1.unique2 = tenk3.unique2
```

Select-All-4, Select-All-5, Select-Count-4, Select-Count-5: Variations of the above queries with 4 and 5 relations in a star query graph shape.

Symmetric Hash Join Expt Queries

3-X, X = 10, 1000
```
SELECT *
FROM hundredk1, hundredk2, hundredk3
WHERE hundredk1.unique2%X = 0 and
hundredk1.unique2 = hundredk2.unique2
and
hundredk1.unique2 = hundredk3.unique2
```

4-X, 5-X, X = 10, 1000
Variations of the above queries with 4 and 5 relations in a star query graph shape.

Index Join Expt Queries

Figure 4: Queries

tuples respectively. The schema of the tables is modeled after the Wisconsin Benchmark schema [4]. We use the following four integer-valued attributes in our experiments: `unique2`, the primary key attribute, and `two`, `four`, and `ten`, attributes that take values 0 to 1, 0 to 3 and 0 to 9 respectively.

3.2 Eddies with Selections

As we mentioned before, selection reordering only makes sense if the query has expensive predicates. To model such expensive predicates, we use a user-defined function called *delay*, that takes two integer arguments, arg_1 and $delay\_value$. This function repeatedly executes an expensive system function (*getimeofday()*), and after time $delay\_value$ *microseconds*, returns the first argument (arg_1) to the caller unmodified.

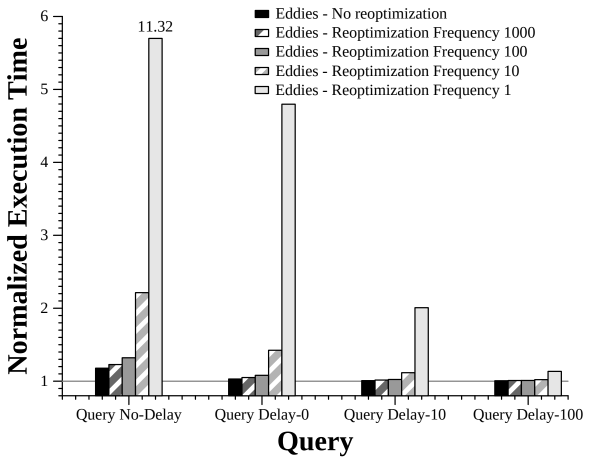

Figure 5: Cost of Adaptivity: Eddies with Selections. The execution times are normalized using the time taken by base PostgreSQL.

For this experiment, we use four queries on the $tenk1$ relation as shown in Figure 4. All these queries return identical answers (every tuple passes all predicates), and as per our philosophy in this study, all selection orderings have identical costs. Figure 5 shows the result of running these queries for base PostgreSQL, and for eddies with selection operators instantiated for each predicate for different reoptimization frequencies. As we can see, even when we are not using the $delay$ function, the cost of eddies is only about twice the cost of base system as long as the reoptimization frequency is less than once every 10 tuples. This is quite remarkable considering how inexpensive these predicates are, and considering that eddies, unlike PostgreSQL, use explicit selection operators to evaluate the predicates. This demonstrates that the extra overhead of routing tuples among the selection operators is not very high. As we can see, if the selection operations are even slightly expensive, this percentage overhead is very small. The overhead also depends on the rate at which reordering is done, but it can be amortized over very few tuples.

3.3 Eddies with Symmetric Hash Joins

For this experiment, we use the six queries shown in Figure 4 that have identical execution cost irrespective of the plan chosen, as a result of the symmetry. Figure 6 shows the result of running these queries for (1) base PostgreSQL, (2) eddies with symmetric hash join operators, and (3) "SHJ"s, a static plan that uses symmetric hash join operators instead of the normal PostgreSQL hash join. Once again, we show results for varying reoptimization frequencies. As we can see, the overhead of using eddies is quite minimal, and in fact, comparing the eddies with "SHJ"s, we see that the overhead is pretty much entirely attributable to use of symmetric hash join operators. A symmetric hash join operator is more expensive than a normal hash join operator because it builds hash tables on both sides of the join. As expected, the overhead is quite high if we try to reoptimize

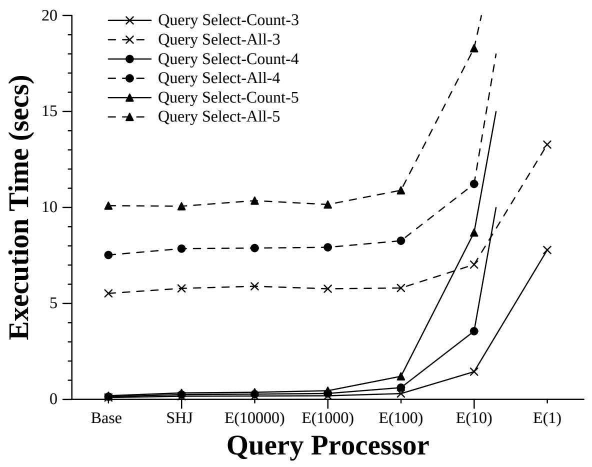

Figure 6: Cost of Adaptivity: Eddies with Symmetric Hash Joins. E(X) denotes eddies with reoptimization frequency X.

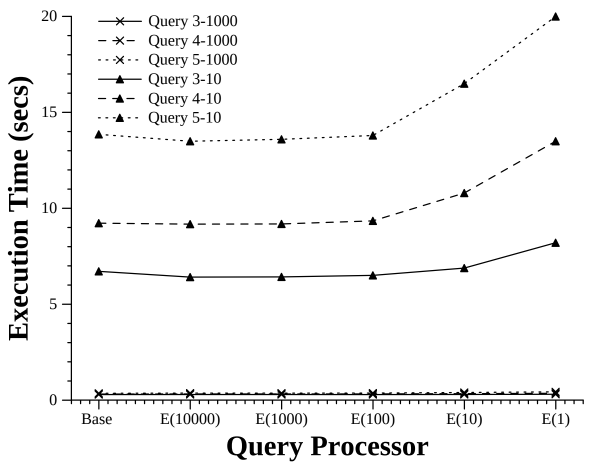

Figure 7: Cost of adaptivity: Eddies with Index Joins

for every tuple, but the reoptimization overhead can be hidden by amortizing over very few tuples.

3.4 Eddies with Index Joins

For this experiment, we created indexes on the relations *hundredk2-5* on the attribute *unique2*, and forced both eddies and the PostgreSQL optimizer to choose index-only plans for all the queries (Figure 4), even though such plans are suboptimal when *X* is 10. Figure 7 shows the results of running these queries for (1) base PostgreSQL, and (2) eddies with index join operators with varying reoptimization frequencies. Once again, we see that the overhead is quite minimal, and can be amortized over very few tuples.

4 Conclusions and Future Work

An eddy is a highly adaptive technique that continuously reoptimizes a query in response to changing runtime conditions. Concern has been expressed about potentially high overhead of this technique. As we demonstrate in this paper, it is possible to implement this technique with modest modification to a traditional database system, providing most of the eddy's proposed flexibility with negligible overhead.

Our focus in this paper was on the overhead of eddies only, for the kind of simple routing policy originally proposed [2]. We are currently studying routing policies in more detail. Our group is also working on building a sharing-oriented continuous query processor over streaming data using the same codebase, and demonstrating that eddies is a viable approach in those kinds of scenarios as well.

Acknowledgments

I would like to thank the members of the TelegraphCQ project at UC Berkeley, in particular, Joe Hellerstein, Mike Franklin and Sailesh Krishnamurthy, for help with this paper. This work was supported by an IBM Fellowship, and by NSF under Grants 0208588 and 0205647.

References

[1] PostgreSQL Data Management System. http://www.postgresql.org.

[2] Ron Avnur and Joe Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[3] Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

[4] David J. DeWitt. The Wisconsin Benchmark: Past, present, and future. In *The Benchmark Handbook Database and Transaction Systems (2nd Edition)*. 1993.

[5] Joe Hellerstein et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 2000.

[6] Sailesh Krishnamurthy et al. TelegraphCQ: An architectural status report. *IEEE Data Engineering Bulletin*, 2003.

[7] Sirish Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[8] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 1993.

[9] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.

[10] Sam Madden, Mehul Shah, Joe Hellerstein, and Vijayshankar Raman. Continously adaptive continous queries over streams. In *SIGMOD*, 2002.

[11] Vijayshankar Raman, Amol Deshpande, and Joe Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[12] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.