Paper Number: 175

Continuous Query Optimization

Ron Avnur Joseph M. Hellerstein
University of California, Berkeley
Berkeley, CA 94720-1776
avnur@cohera.com, jmh@cs.berkeley.edu

Abstract

In large federated and shared-nothing databases, resources can exhibit widely fluctuating characteristics. Assumptions made at the time a query is submitted will rarely hold throughout the duration of query processing. As a result, traditional static query optimization and execution techniques are ineffective in these environments.

In this paper we introduce a query processing mechanism called an *eddy*, which continuously reorders operators in a query plan as it runs. We characterize the *moments of symmetry* during which pipelined joins can be easily reordered, and the *synchronization barriers* that require inputs from different sources to be coordinated. By combining eddies with join algorithms appropriate to large-scale environments, we merge the optimization and execution phases of query processing, allowing each tuple to have a flexible ordering of the query operators. This flexibility is controlled by a combination of fluid dynamics and a simple learning algorithm. Our initial implementation demonstrates promising results, with eddies performing nearly as well as a static optimizer/executor in static scenarios, and adapting to run-time changes in the execution environment.

1 Introduction

There is increasing interest in query engines that run at unprecedented scale, both for widely-distributed information resources, and for massively parallel database systems. We are building a system called Telegraph, a Global DBMS that is intended to run queries over all the data available on line [HBF99]. A key requirement of a large-scale system like Telegraph is that it function robustly in an unpredictable and constantly fluctuating environment. This unpredictability is endemic in large scale systems, because of increased complexity in a number of dimensions:

- **Hardware and Workload Complexity:** In wide-area environments, variabilities are commonly observable in the bursty performance of servers and networks [AFTU96, UFA98]; large servers and networks tend to serve large communities of users, whose aggregate behavior can be hard to predict, and the hardware mix in the wide area is typically quite heterogeneous. Large clusters of computers can exhibit similar performance variations, due to a mix of user requests and heterogeneous hardware evolution. Even in totally homogeneous environments, hardware performance can be unpredictable: for example, the outer tracks of a disk can exhibit almost twice the bandwidth of inner tracks [Met97].
- **Data Complexity:** Selectivity estimation for static alphanumeric data sets is fairly well understood, and there has been some initial work on estimating statistical properties of static sets of data with complex types [Aok99] and methods [BVO97, BO99]. But federated data often comes without any statistical summaries, and complex non-alphanumeric data types are now widely in use both in object-relational databases and on the web. In these scenarios – and even in traditional static relational databases – selectivity estimates are often quite inaccurate.

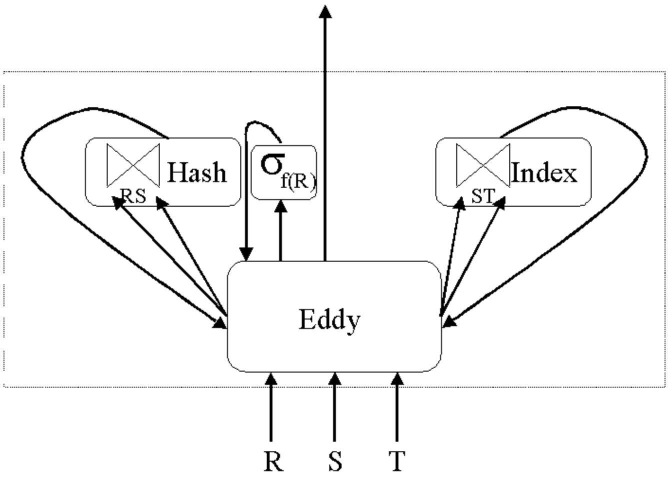

Figure 1: An eddy in a query pipeline. Data flows into the eddy from input relations R, S and T. The eddy routes tuples to operators; the operators run as independent threads, returning tuples to the eddy. The eddy sends a tuple to the output only when it has been handled by all the operators. The eddy can adaptively choose an independent order for routing each tuple through the operators.

- **User Interface Complexity:** In large-scale systems, many queries can run for a very long time. As a result, there is interest in Online Aggregation and other techniques that allow users to "Control" properties of queries while they execute, based on refining approximate results [HAC+99, RRH99].

For all of these reasons, we expect query processing parameters to change significantly over time in Telegraph, typically many times during the execution of a single query. As a result, it is not appropriate to use the traditional architecture of optimizing a query and then executing a static query plan: this approach does not adapt to intra-query fluctuations. Instead, for these environments we want query execution plans to be reoptimized regularly during the course of query processing, allowing the system to adapt dynamically to fluctuations in computing resources, data characteristics, and user preferences.

In this paper we present a query processing mechanism called an *eddy*, which continuously reorders the application of pipelined operators in a query plan, on a tuple-by-tuple basis. An eddy is an n-ary tuple router interposed between n data sources and a set of query processing operators; the eddy encapsulates the ordering of the operators by routing tuples through them dynamically (Figure 1). Because the eddy observes tuples entering and exiting the operators, it can adaptively change its routing to effect different orders of the pipelined operators. In this paper we present initial experimental results demonstrating the viability of eddies: they can indeed reorder effectively in the face of changing selectivities and costs, and provide benefits in the case of delayed data sources as well. Our results demonstrate that the eddy framework is a promising avenue of research for adaptive and online query processing.

Reoptimizing a query execution pipeline on the fly requires significant care in maintaining query execution state. In this paper we highlight query processing stages called *moments of symmetry*, during which operators can be easily reordered. We also describe *synchronization barriers* in certain join algorithms that can restrict performance to the rate of the slower input. Join algorithms with frequent moments of symmetry and adaptive or non-existent barriers are thus especially attractive in the Telegraph environment. We observe that the Ripple Join family [HH99] provides efficiency, frequent moments of symmetry and adaptive or nonexistent barriers for equijoins and non-equijoins alike.

The eddy architecture is quite simple, obviating the need for traditional cost and selectivity estimation, and simplifying the logic of plan enumeration. Eddies represent our first step in a larger attempt to do away with traditional optimizers entirely, in the hope of providing both run-time adaptivity and significant reduction in code complexity. In this first paper we focus on continuous operator reordering in a single-site query processor; we leave issues of operator selection and parallelism to our discussion of future work.

1.1 Run-Time Fluctuations

Three properties can vary during query processing: the costs of operators, the selectivities of operators, and the rates at which tuples arrive from the inputs. The first and third issues are well discussed in the literature [AFTU96, KD98, UFA98, IFF+99]. A standard example of a variable-cost operator is an index lookup in a network service, e.g. a web form or an LDAP server that serves a large community of users. Burstiness in utilization (and hence performance) is now common in wide-area services, and may become more common in cluster systems as they "scale out" to thousands of nodes or more [Bar99, Gra99]. Run-time variations in selectivity have not been widely discussed in the literature. Selectivites can indeed change mid-query, and our techniques for run-time reoptimization of pipelined operators can take advantage of this fact to change plans as appropriate during the course of query execution.

Changes in selectivity commonly arise due to correlations between the order of tuple delivery and the predicates. For example, consider an employee table clustered by age, and a selection `salary > 100000`; age and salary are often strongly correlated. Initially the selection will filter out most of the tuples it sees, but that selectivity rate will change as older employees are scanned. The selectivity over time can also be tied to performance fluctuations: e.g., in a parallel DBMS clustered relations are often horizontally partitioned across disks, and the rate of production from various partitions – and hence from different value ranges – may change over time depending on performance characteristics and utilization of the different disks. Finally, Online Aggregation systems explicitly allow users to control the order in which tuples are delivered based on data preferences [RRH99]; this can have a similar effect on query predicates that are correlated with the user-controlled preferences.

1.2 Architectural Assumptions

Telegraph is intended to efficiently and flexibly provide both distributed query processing across sites in the wide area, and parallel query processing in a large shared-nothing cluster. In this paper we narrow our focus somewhat to concentrate on the initial, already difficult problem of run-time operator reordering in a single-site query executor; that is, changing the effective order or "shape" of a pipelined query plan tree in the face of changes in performance.

We make some significant simplifying assumptions in this initial paper, to allow us to focus carefully on the problem run-time reordering. We assume that some initial query plan tree will be constructed during parsing by a naive *pre-optimizer*. This optimizer need not exercise much judgement since we will be reordering the plan tree on the fly. However by constructing a query plan it must choose a spanning tree of the query graph (i.e. a set of table-pairs to join) [KBZ86], and algorithms for each of the joins. We will return to the choice of join algorithms in Section 2, and defer to Section 6 the discussion of changing the spanning tree and join algorithms during processing.

We study a standard single-node object-relational query processing system, with the added capability of opening scans and indexes from external data sets. This is becoming a very common base architecture, available in a number of the commercial object-relational systems (e.g., IBM DB2 UDB [RPK+99], Informix Dynamic Server UDO [SBH98]) and in federated database systems (e.g., Cohera [HSC99]). For uniformity, we refer to these non-resident tables as *external tables*. External tables are now a practical mechanism for integrating data from legacy systems of various kinds. In one typical application scenario, a remote SAP R/3 application could expose a set of business objects as flat XML structures via a system like webMethods B2B [Tho99]; the DBMS schema would have metadata naming this set as a table and modeling its attributes as columns, and would provide an XML "gateway" or "wrapper" to interpret the data as it streams in. This external table could then be joined by the DBMS with local tables and other external tables. This scenario has also been a topic of study in other research on heterogeneous databases and data integration, e.g., [IFF+99, HKWY97, GMPQ+97, ACPS96]. Unlike some of the prior work, we make no assumptions limiting the scale of external sources, which may be arbitrarily large. Note that external tables present many of the dynamic challenges described above: they can reside over a wide-area network, face bursty utilization, and offer very minimal information on costs and statistical properties.

In a large and unpredictable environment, early results are often a required feature, since the time to run a query to completion may be enormous. As a result, we focus on reordering the operators within a pipeline; for the duration of the paper, when we speak of query plans we refer implicitly to pipelined plans (or subplans). In Section 5 we outline synergies and tensions between this approach and prior work on inter-pipeline run-time optimization.

1.3 Structure of the Paper

Before introducing eddies, in Section 2 we discuss the properties of query processing algorithms that allow (or disallow) them to be frequently reordered. We then present the eddy architecture, and describe how it allows for extreme flexibility in operator ordering (Section 3). Section 4 discusses the policies for controlling tuple flow in an eddy. A variety of experiments in Section 4 illustrate the robustness of eddies in both static and dynamic environments, and raise some questions for future work. We survey related work in Section 5, and in Section 6 lay out a research program to carry this work forward.

2 Reorderability of Plans

A basic challenge of run-time reoptimization is to reorder pipelined query processing operators while they are in flight. To change a query plan on the fly, a great deal of state in the various operators has to be considered, and arbitrary changes can require significant processing and code complexity to guarantee correct results. For example, the state maintained by an operator like hybrid hash join [DKO+84] can grow as large as the size of an input relation, and require modification or recomputation if the plan is reordered while the state is being constructed.

By constraining the scenarios in which we reorder operators, we can keep this work to a minimum. Before describing eddies, we study the state management of query processing algorithms; this discussion motivates the eddy design, and forms the basis of our approach for reoptimizing cheaply and continuously. As a philosophy, *we favor adaptivity over best-case performance*. In a highly variable environment, the best-case scenario rarely exists for a significant length of time. So we will sacrifice marginal improvements in idealized query processing algorithms when they prevent frequent, efficient reoptimization.

2.1 Single-Table Operators

We begin by studying the simple scenario of reordering unary (single-table) operators during query execution. The standard unary operators in an object-relational system include selections, projections and output transformation ("Select list") expressions, all of which are processed in a simple manner: they repeatedly fetch a tuple from their input, apply some logic to that tuple (possibly including expensive user-defined functions), and then perhaps output a tuple. This "tuple-at-a-time" processing maintains no state across tuples: the values of one input tuple are not considered when processing subsequent tuples. Since these operators are stateless, run-time reordering can be done trivially – each operator can remain encapsulated, and unaware of the change in ordering[1]. Not all operators can commute, of course, but constraints on legal orderings are orthogonal to our concerns in this section about state management. Ordering constraints do not complicate the task of swapping any two operators that are indeed commutable.

[1]As a detail, note that the types of tuples passed into an operator should not change on the fly. This is relatively easy to ensure in a relational context, without imposing undue constraints on reordering. All operators in a query should share a common namespace for column variables (e.g. "column 3 of table R"), and for expressions ("query expression #5"). Contrast this with the alternative approach, where column variables are parameterized by input relation (e.g. "column 3 of my left input"). This latter implementation is used in POSTGRES, and complicates even static query plan modifications, like those of [Hel98].

Even the stateful unary operators like sorting and grouping can be easily reordered, subject to constraints on legal orderings. Both sorting and grouping operators consume an entire input relation before producing any output. While in the midst of reading from their inputs they are insensitive to changes in subsequent input; no relationship is assumed between previously-seen and as-yet-unseen input records to be sorted or grouped. Of course for correctness some reorderings are not allowed; for example, one cannot in general commute a grouping operator with a selection if the grouping operator computes an aggregate. But again, this is an issue about semantically correct reorderings, and is orthogonal to our discussion. We will return to the issue of the desirability of these operators in Section 2.4.

In short, when unary operators can be legally reordered, the logic of doing so is trivial since no state changes are required. We proceed to consider reordering joins, which we will see are significantly more complex.

2.2 Binary Operators: Synchronization Barriers

Binary operators like joins often capture significant state. A particular form of state used in such operators relates to the interleaving of requests for tuples from different inputs. Some join algorithms must exercise significant control over the order in which they consume tuples from one input or the other, while other join algorithms can consume tuples from either input with arbitrary interleaving.

As an example, consider the case of a merge join on two sorted, duplicate-free inputs. During the processing of a merge join, the next tuple is always consumed from the relation whose last tuple had the lower value. This significantly constrains the order in which tuples can be consumed: as an extreme example, consider the case of a slowly-delivered external relation slowlow with many low values in its join column, and a high-bandwidth but large local relation fasthi with only high values in its join column – the processing of fasthi is postponed for a long time while consuming many tuples from slowlow. Using terminology from parallel programming, we describe this phenomenon as a *synchronization barrier*: one table-scan must wait until the other table-scan produces a value larger than any seen before.

In general, barriers limit concurrency – and hence performance – when two tasks take different amounts of time to complete (i.e., to "arrive" at the barrier). Recall that concurrency arises even in single-site query engines, which can simultaneously carry out network I/O, disk I/O, and computation. Thus it is desirable to minimize the overhead of synchronization barriers in a dynamic (or even static but heterogeneous) performance environment. Two issues affect the overhead of barriers in a plan: the frequency of barriers, and the gap between arrival times of the two inputs at the barrier. We will see in upcoming discussion that barriers can often be avoided or at least minimized by using appropriate join algorithms.

2.3 Binary Operators: Moments of Symmetry

Note that the synchronization barrier in merge join is stated in an order-independent manner: it does not distinguish between the inputs based on any property other than the data they deliver. Thus merge join is often described as a symmetric operator, since its two inputs are treated uniformly[2]. This is not the case for many other join algorithms. Consider the traditional nested-loops join, for example. The "outer" relation in a nested-loops join is synchronized with the "inner" relation, but not vice versa: after each tuple (or block of tuples) is consumed from the outer relation, a barrier is set until a full scan of the inner is completed. For asymmetric operators like nested-loops join, performance benefits can often be obtained by reordering the inputs.

When a join algorithm reaches a barrier, it has declared the end of a scheduling dependency between its two input relations. In such cases, the order of the inputs to the join can often be changed without modifying any state in the join; when this is true, we refer to the barrier as a *moment of symmetry*. Let us return to the example of a nested-loops join, with outer relation R and inner relation S. At a barrier, the join has completed a full inner loop, having joined each tuple in a subset of R with every tuple in S. Reordering the

[2] If there are duplicates in a merge join, the duplicates are handled by an asymmetric but usually small nested loop. For ease of exposition, we can ignore this detail in our discussion.

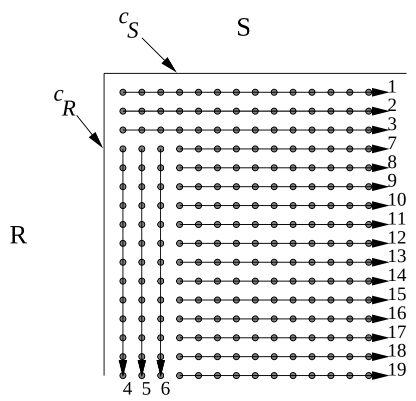

Figure 2: Tuples generated by a nested-loops join, reordered at two moments of symmetry. Each axis represents the tuples of the corresponding relation, in the order they are delivered by an access method. The dots represent tuples generated by the join, some of which may be eliminated by the join predicate. The numbers correspond to the barriers reached, in order. c_R and c_S are the cursor positions maintained by the corresponding inputs at the time of the reorderings.

inputs at this point can be done without affecting the join algorithm, as long as the iterator producing R notes its current cursor position c_R. In that case, the new "outer" loop on S begins rescanning by fetching the first tuple of S, and R is scanned from c_R to the end. This can be repeated indefinitely, joining S tuples with all tuples in R from position c_R to the end. Alternatively, at the end of some loop over R (during a moment of symmetry), the order of inputs can be swapped again by remembering the current position of S, and repeatedly joining the next tuple in R (starting at c_R) with tuples from S between c_S and the end. Figure 2 depicts this scenario, with two changes of ordering. Some operators like the pipelined hash join of [WA91] have no barriers whatsoever. These operators are in constant symmetry, since the processing of the two inputs is totally decoupled.

Moments of symmetry allow reordering of the inputs to a single binary operator. To generalize this somewhat, note that since joins commute, a tree of $n-1$ binary joins can be viewed as a single n-ary join. One could easily implement a doubly-nested-loops join operator over relations R, S and T, and it would have moments of complete symmetry at the end of each loop of S. At that point, all three inputs could be reordered (say to T then R then S) with a straightforward extension to the discussion above: a cursor would be recorded for each input, and each loop would go from the recorded cursor position to the end of the input.

The same effect can be obtained in a binary implementation with two operators, by swapping the positions of binary operators: effectively the plan tree transformation would go in steps, from $(R \bowtie_1 S) \bowtie_2 T$ to $(R \bowtie_2 T) \bowtie_1 S$ and then to $(T \bowtie_2 R) \bowtie_1 S$. This approach treats an operator and one of its inputs as a unit (e.g., the unit $[\bowtie_2 T]$), and swaps units; the idea has been used previously in static query optimization schemes [IK84, KBZ86, Hel98]. Viewing the situation in this manner, we can naturally consider reordering multiple joins and their inputs, even if the join algorithms are different. In our query $(R \bowtie_1 S) \bowtie_2 T$, we need $[\bowtie_1 S]$ and $[\bowtie_2 T]$ to be mutually commutative, but do not require them to be the same join algorithm. We discuss the commutativity of join algorithms further in Section 2.4.3.

Note that the combination of commutativity and moments of symmetry allows for very aggressive reordering of a plan tree. A single n-ary operator representing a reorderable plan tree is therefore an attractive abstraction, since it encapsulates any ordering that may be subject to change. We will exploit this abstraction directly, by interposing an n-ary tuple router (an "eddy") between the input tables and the join operators.

2.4 Reorderability of Join Algorithms

There are a number of join algorithms in the literature, including the ones commonly used in today's systems, and relevant alternatives proposed for "online" [HH99] and adaptive [UF99, IFF$^+$99] processing. Some of the prior work on querying external tables focused exclusively on the pipelined hash join [WA91], and found it to be very effective; in prior work this was attributed to its lack of barriers, without leveraging its constant symmetry for reordering. Hash joins are not always appropriate, however, since they do not take advantage of indexes and only work for equijoin predicates. In this section we consider many join algorithms in light of the discussion above, highlighting a general class of algorithms that allow for flexible run-time reordering, and adaptive or non-existent barriers.

2.4.1 Blocking vs. Pipelining

One can coarsely divide standard query processing methods into *blocking* and *pipelining* algorithms. Blocking algorithms do not produce any output until they consume one or both input relations; the most common binary example is the hybrid hash join[3]. Pipelining algorithms can produce output while concurrently reading from their inputs. Examples of pipelining joins include merge joins, as well as the Ripple Join family [HH99] that generalizes nested-loops joins, pipelining hash joins [WA91], and variants of the two. Blocks in a pipeline represent barriers between an operator and all the operators that follow it in the data flow. This barrier is a (trivial) moment of symmetry for the subsequent operators, since they cannot begin until the barrier is crossed. Since the operators after the block have not begun processing, the symmetry after a block is analogous to statically optimizing a (sub-)query before running it; this problem is treated in [KD98, IFF$^+$99]. We will focus here strictly on pipelining algorithms, which can be aggressively reordered without long waits for the completions of barriers (like the barrier of a blocking operator finishing.) Our work here can be combined with cross-pipeline optimizations like those of [KD98, IFF$^+$99], as we discuss in Section 5.

2.4.2 Joins and Indexes

Nested-loops joins can take advantage of indexes on the inner relation, resulting in a fairly efficient pipelining join algorithm. An index nested-loops join (henceforth an "index join") is inherently asymmetric, since one input relation has been pre-indexed. Even when indexes exist on both inputs, changing the choice of inner and outer relation "on the fly" is problematic[4]. Hence for the purposes of reordering, it is simpler to think of an index join as a kind of unary selection operator on the unindexed input. The only distinction between an index join and a selection is that – with respect to the unindexed relation – the selectivity of the join node may be greater than 1. Although one cannot swap the inputs to a single nested-loops join, one can reorder a nested-loops join and its indexed relation as a unit among other operators in a plan tree. Note that the logic for indexes can be applied to external tables that require bindings to be passed; such tables may be gateways to, e.g., web pages with forms, GIS index systems, LDAP servers and so on [HKWY97, FMLS99].

2.4.3 Physical Properties, Join Predicates and Commutativity

Clearly, a pre-optimizer's choice of an index join algorithm constrains the possible join orderings. In the n-ary join view, an ordering constraint must be imposed so that the unindexed join input is ordered before (but not necessarily directly before) the indexed input. This constraint arises because of a *physical property* of an input relation: indexes can be probed but not scanned, and hence cannot appear before their corresponding

[3] Note that sort-merge join can be modeled as two blocking unary sorts, followed by one non-blocking binary merge operator. A similar description can be made for grace hash join [Sha86]. More generally, binary operators that are fully symmetric while reading their inputs can be decomposed into two independent unary operators followed by a binary operator.

[4] In unclustered indexes, the index ordering is not the same as the scan ordering. Thus after a reordering of the inputs it is difficult to ensure that – using the terminology of Section 2.3 – lookups on the index of the new "inner" relation R produce only tuples between c_R and the end of R. When this is not ensured, care must be taken to eliminate incorrect duplicates from the result.

Algorithm	Moments of Symmetry	Input Barriers	Order Constraints
Hybrid Hash Join	none	none	no $\times$-products
Merge Join	end of each nested loop over duplicates	data dependent	inputs must be sorted, no $\times$-products
Nested Loops	end of each inner loop	end of each inner loop	none
Block Ripple	at each corner	at each corner, but corners chosen adaptively	none
Index Join	end of each index probe	end of each index probe	outer precedes inner, no $\times$-products
Hash Ripple (+ pipelined hash join, X-join)	after each tuple	none	no $\times$-products

Table 1: Reorderability Properties of Various Join Algorithms

probing tables. Similar but more extensive constraints can arise in preserving the ordered inputs to a merge join.

The applicability of certain join algorithms raises additional constraints. Many join algorithms work only for equijoins, and will not work on other joins like Cartesian products. Such algorithms constrain reorderings on the plan tree as well, since they require all the relations mentioned in their equijoin predicates to always be executed before them on each tuple. In this paper, we consider ordering constraints to be an inviolable aspect of a plan tree, and we ensure that they always hold. In Section 6 we sketch some initial ideas on relaxing this requirement, by simultaneously considering multiple join algorithms and query graph spanning trees.

2.4.4 Join Algorithms and Reordering

In order for an eddy to be most effective, we favor join algorithms with frequent moments of symmetry, adaptive or non-existent barriers, and minimal ordering constraints: these algorithms offer the most opportunities for reoptimization. Table 1 summaries the salient properties of a variety of join algorithms. Our desire to avoid blocking rules out the use of hybrid hash join, and our desire to minimize ordering constraints and barriers excludes merge joins. Nested loops joins have infrequent moments of symmetry and imbalanced barriers, making them undesirable as well.

All the remaining algorithms we consider are based on frequently-symmetric versions of traditional iteration, hashing and indexing schemes, i.e. the Ripple Joins [HH99]. Note that the original pipelined hash join of [WA91] is a constrained version of the hash ripple join. The external hashing extensions of [UF99, IFF+99] are applicable to the variable-rate hash ripple join, and [HH99] captures index joins as a special case as well. For non-equijoins, the block ripple join algorithm is effective, having fairly frequent moments of symmetry, particularly at the beginning of processing [HH99]. Figure 3 illustrates block, index and hash ripple joins; the reader is referred to [HH99, IFF+99, UF99] for detailed discussions of these algorithms and their variants. These algorithms are adaptive without sacrificing much performance: [UF99] and [IFF+99] demonstrate scalable versions of hash ripple join that perform competitively with hybrid hash join in the static case; [HH99] shows that while block ripple join can be less efficient than nested-loops join, it arrives at moments of symmetry much more frequently than nested-loops joins, especially in early stages of processing. Appendix A discusses the memory overheads of these adaptive algorithms, which can be somewhat larger than standard join algorithms.

Ripple joins have moments of symmetry at each "corner" of a rectangular ripple in Figure 3, i.e., whenever a prefix of the input stream R has been joined with all tuples in a prefix of input stream S and vice versa. For hash ripple joins (like the pipelined hash join) and index joins, this scenario occurs between each consecutive tuple consumed from a scanned input. Thus ripple joins offer very frequent moments of symmetry.

Ripple joins are attractive with respect to barriers as well. Ripple joins were designed to allow changing

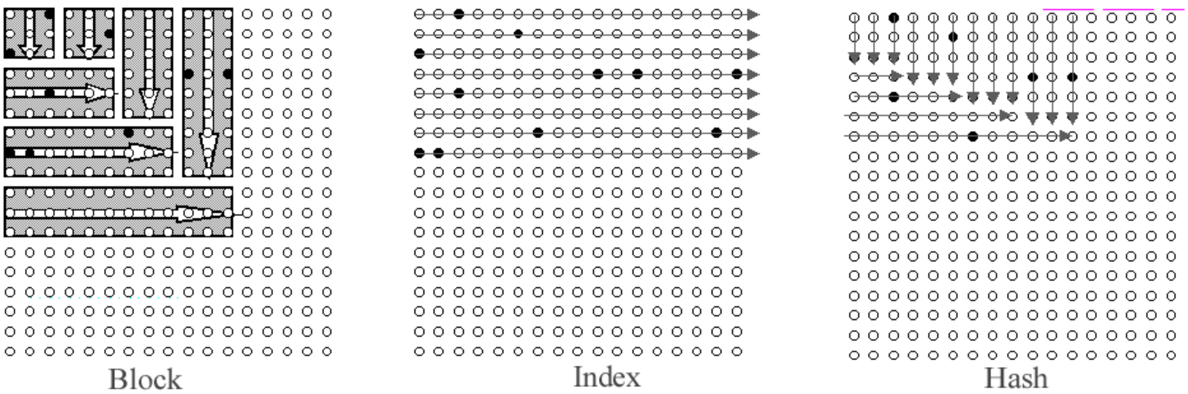

Figure 3: Tuples generated by block ripple join, index join, and hash ripple. In the hash ripple diagram, one relation arrives 3x faster than the other. Each axis represents the tuples of an input relation, in the order they are delivered by an access method. In the block ripple join, all tuples are generated by the join, but some may be eliminated by the join predicate. In the index and hash ripple joins, the only tuples generated by the join are those which satisfy the join predicates; these are indicated by black dots. The arrows for index and hash ripple join represent the *logical* portion of the cross-product space checked so far; the index and hash tables only expend work on tuples that satisfy the join predicates.

rates for each input; this was originally used to proactively expend more processing on the input relation with more statistical influence on intermediate results. However, the same mechanism allows reactive adaptivity in the wide-area scenario: a barrier is reached at each corner, and the next corner can adaptively reflect the relative rates of the two inputs. For the block ripple join, the next corner is chosen upon reaching the previous corner; this can be done adaptively to reflect the relative rates of the two inputs over time.

The ripple join family offers attractive adaptivity features at a modest overhead in performance and memory footprint. Hence they fit well with our philosophy of sacrificing marginal speed for adaptability, and we focus on these algorithms in Telegraph. It is true that additional memory requirements can be problematic in boundary cases, sometimes adding additional passes over the data to accomodate overflow. We believe that technology trends justify expending memory to maximize adaptability, but as we note in Section 5 and Appendix A, one can mix blocking operators with eddy pipelines if desired.

3 Rivers and Eddies

The above discussion allows us to consider easily reordering query plans at moments of symmetry. In this section we proceed to describe the eddy mechanism for implementing reordering in a natural manner during query processing. The techniques we proceed to describe can be used with any operators, but algorithms with frequent moments of symmetry allow for more frequent reoptimization. Before discussing eddies, we first introduce our basic query processing environment.

3.1 River

We have implemented eddies in the context of River [ADAT+99]. River is a shared-nothing parallel query processing framework that can dynamically adapt to fluctuations in performance and workload. It has been used to robustly produce near-record performance on I/O-intensive benchmarks like parallel sorting and hash joins, despite heterogeneities and dynamic variability in hardware and workloads across machines in a cluster. For more details on River's adaptivity and parallelism features, the interested reader is referred to the original paper on the topic [ADAT+99]. In Telegraph, we intend to leverage the adaptability of River to allow for dynamic shifting of load (both query processing and data delivery) in a shared-nothing parallel environment. But in this paper we restrict ourselves to basic (single-site) features of eddies; discussions of eddies in parallel rivers are deferred to Section 6.

Since we do not discuss parallelism here, a very simple overview of the River framework suffices. River is a dataflow query engine, analogous in many ways to Gamma [DGS+90], Volcano [Gra90] and many commercial parallel database engines, in which "iterator"-style *modules* (query operators) communicate via a fixed dataflow graph (a query plan). Each module runs as an independent thread, and the edges in the graph correspond to finite message queues. When a producer and consumer run at differing rates, the faster thread may block on the queue waiting for the slower thread to catch up. River also provides a scripting syntax and graphical user interface for specifying query plans; we currently use this feature to perform pre-optimization of queries by hand for experimentation. Like the modified Predator architecture used in Query Scrambling [UFA98], River is multi-threaded and can exploit barrier-free algorithms by reading from various inputs at independent rates. The River implementation we used derives from the work on Now-Sort [ADADC+97], and features efficient I/O and scheduling mechanisms including pre-fetching scans, avoidance of operating system buffering, and high-performance user-level networking.

3.1.1 Pre-Optimization

Although we will use eddies to reorder tables among joins, a heuristic pre-optimizer must choose how to initially pair off relations into joins, with the constraint that each relation participates in only one join. This corresponds to choosing a spanning tree of a query graph, in which nodes represent relations and edges represent binary joins [KBZ86]. One reasonable heuristic for picking a spanning tree forms a chain of cartesian products across any tables known to be very small (to handle "star schemas" when base-table cardinality statistics are available); it then picks arbitrary equijoin edges (on the assumption that they are relatively low selectivity), followed by as many arbitrary non-equijoin edges as required to complete a spanning tree.

Given a spanning tree of the query graph, the pre-optimizer needs to choose join algorithms for each edge. Along each equijoin edge it can use either an index join if an index is available, or a hash ripple join. Along each non-equijoin edge it can use a block ripple join.

The focus of our work to date has been on validating the design of an eddy, and its ability to reorder operators in a pre-specified spanning tree, with pre-specified join algorithms. We have not yet experimented with the sensitivity of the system to choices made during pre-optimization. Rather than validate the heuristics above, however, we are focusing our energy on extending eddies with the capability of adaptively modifying the spanning tree and join algorithms during processing. In Section 6 we discuss our initial ideas in this direction. In the current paper, we assume that a heuristic pre-optimizer makes these decisions; in our experiments we perform this heuristic by hand.

3.2 An Eddy in the River

An eddy is implemented via a module in a river containing an arbitrary number of input relations, a number of participating unary and binary modules, and a single output relation (Figure 1)[5]. An eddy encapsulates the scheduling of its participating operators; tuples entering the eddy can flow through its operators in a variety of orders.

In essence, an eddy explicitly merges multiple unary and binary operators into a single n-ary operator within a query plan, based on the intuition from Section 2.3 that symmetries can be easily captured in an n-ary operator. An eddy module maintains a small fixed-sized buffer of tuples that are to be processed by one or more operators. Each operator participating in the eddy has one or two inputs that are fed tuples by the eddy, and an output stream that returns tuples to the eddy. Eddies are so named because of this circular data flow within a river.

[5]Nothing prevents the use of n-ary operators with $n > 2$ in an eddy, but since implementations of these are atypical in database query processing we do not discuss them here.

A tuple entering an eddy is associated with a tuple descriptor containing a vector of *Ready* bits and *Done* bits, which indicate respectively those operators that are elgibile to process the tuple, and those that have already processed the tuple. The eddy module ships a tuple only to operators for which the corresponding Ready bit turned on. After processing the tuple, the operator returns it to the eddy, and the corresponding Done bit is turned on. If all the Done bits are on, the tuple is sent to the eddy's output; otherwise it is sent to another eligible operator for continued processing.

When an eddy receives a tuple from one of its inputs, it zeroes the Done bits, and sets the Ready bits appropriately. In the simple case, the eddy sets all Ready bits on, signifying that any ordering of the operators is acceptable. It is possible for ordering constraints to be placed on the operators; in such cases, the eddy is parameterized to turn on only the Ready bits corresponding to operators that can be executed initially. When an operator returns a tuple to the eddy, the eddy turns on the Ready bit of any operator eligible to process the tuple. Binary operators generate output tuples that correspond to combinations of input tuples; in these cases, the Done bits and Ready bits of the two input tuples are ORed. In this manner an eddy preserves the ordering constraints while maximizing opportunities for tuples to follow different possible orderings of the operators.

Two properties of eddies merit comment. First, note that eddies represent the full class of bushy trees corresponding to the set of join nodes – it is possible, for instance, that two pairs of tuples are combined independently by two different join modules, and then routed to a third join to perform the 4-way concatenation of the two binary records. Second, note that eddies do not constrain reordering to moments of symmetry across the eddy as a whole. A given operator must carefully refrain from fetching tuples from certain inputs until its next moment of symmetry – e.g., a nested-loops join would not fetch a new tuple from the current outer relation until it finished rescanning the inner. But there is no requirement that *all* operators in the eddy be at a moment of symmetry when this occurs; just the operator that is fetching a new tuple. Thus eddies are quite flexible both in the shapes of trees they can generate, and in the scenarios in which they can logically reorder operators.

4 Routing Tuples in Eddies

An eddy module paces and directs the flow of tuples from the inputs through the various operators to the output. This provides the essential flexibility to allow each tuple to be routed individually through the operators. The routing policy used in the eddy determines the efficiency of the system. In this section we study these policies in detail; the policies we present here show promise, but we believe that this is a rich area for future study. We outline some of the remaining questions in Section 6.

An eddy's tuple buffer is implemented as a priority queue with a flexible prioritization scheme. An operator is always given the highest-priority tuple in the buffer that has the corresponding Ready bit set. For simplicity, we start by considering a very simple priority scheme: tuples enter the eddy with low priority, and when they are returned to the eddy from an operator they are given high priority. This simple priority scheme ensures that tuples flow completely through the eddy before new tuples are consumed from the inputs, ensuring that the eddy does not become "clogged" with new tuples.

4.1 Experimental Setup

In order to illustrate how eddies work, we present some initial experiments in this section; we pause briefly here to describe our experimental setup. All our experiments were run on a single-processor Sun Ultra-1 workstation running Solaris 2.6, with 160 MB of RAM. We used the Euphrates implementation of River [ADAT+99]. We synthetically generated relations as in Table 2, with 100 byte tuples in each relation.

To allow us to experiment with costs and selectivities of selections, our selection modules are (artificially) implemented as spin loops corresponding to their relative costs, followed by a randomized selection decision with the appropriate selectivity. We will describe the relative costs of selections in terms of abstract "delay units"; for studying optimization, the absolute number of cycles through a spin loop are irrelevant. We

Table	Cardinality	values in column a
R	10,000	500 - 5500
S	80,000	0 - 5000
T	10,000	N/A
U	50,000	N/A

Table 2: Cardinalities of tables; values are uniformly distributed.

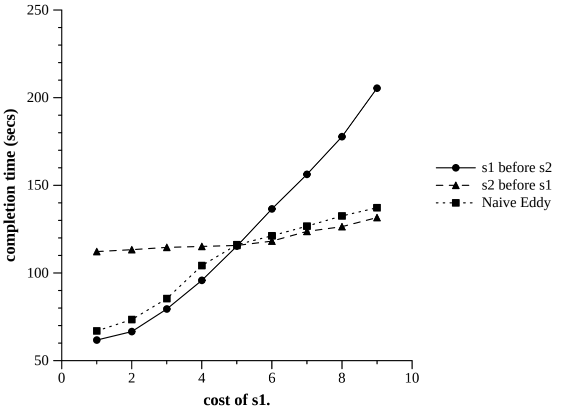

Figure 4: Performance of two 50% selections, $s2$ has cost 5, $s1$ varies across runs.

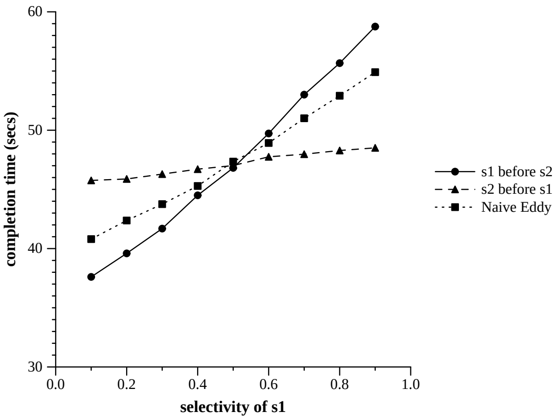

Figure 5: Performance of two selections of cost 5, $s2$ has 50% selectivity, $s1$ varies across runs.

implemented the simplest version of hash ripple join, identical to the original pipelining hash join [WA91]; our implementation here does not exert any statistically-motivated control over disk resource consumption (as in [HH99]). We simulated index joins by doing random I/Os within a file, returning on average the number of matches corresponding to a pre-programmed selectivity. The filesystem cache was allowed to absorb some of the index I/Os after warming up.

The eddy's tuple buffer was implemented using River's efficient *Distributed Queue* mechanism, in order to enable future research on parallel eddies. The ramification of this implementation is that each batch of tuples entering the eddy causes a Myrinet network card to be notified; the card notes that the tuple's destination is local, and returns a notification. This logic is done in a very lightweight user-level network protocol called Active Messages [vECG+92]; it does not entail a context switch to kernel mode, but it does cause a small measurable effect on performance. In order to mask this implementation detail in our study, we simulate static plans by using eddies that enforce a static ordering on tuples (via setting Ready bits in the correct order). An implementation that avoided the network card would thus benefit eddies and static plans by the same constant factor.

4.2 Naive Eddy: Fluid Dynamics and Operator Costs

To illustrate how an eddy works, we consider a very simple single-table query with two expensive selection predicates, under the traditional assumption that no performance or selectivity properties change during execution. Our SQL query is simply the following:

```
SELECT  *
  FROM  U
 WHERE  s1() AND s2();
```

In our first experiment, we wish to see how well an eddy can account for differences in costs among operators. We run the query multiple times, always setting the cost of $s2$ to 5 delay units, and the selectivities of both

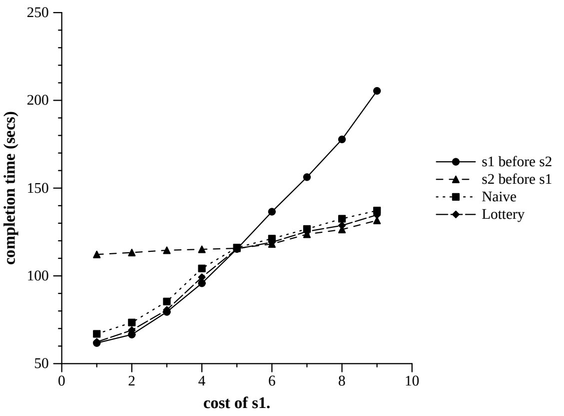

Figure 6: Performance of two 50% selections, $s2$ has cost 5, $s1$ varies across runs.

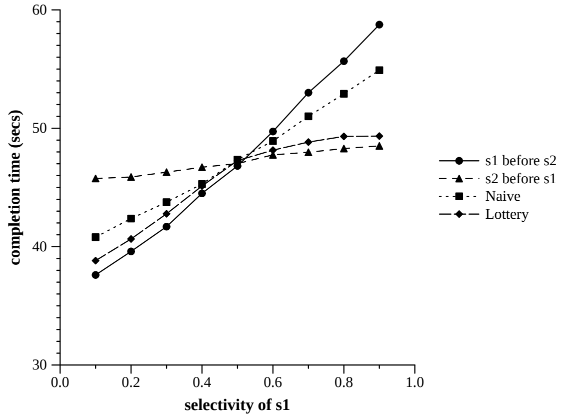

Figure 7: Performance of two selections of cost 5, $s2$ has 50% selectivity, $s1$ varies across runs.

selections to 50%. In each run we use a different cost for $s1$, varying it between 1 and 9 delay units across runs. We compare an eddy of the two selections against both possible static orderings of the two selections. One might imagine that the flexible routing in the eddy would deliver tuples to the two selections equally: half the tuples would flow to $s1$ before $s2$, and half to $s2$ before $s1$, resulting in middling performance over all. Figure 4 shows that this is not the case: the eddy nearly matches the better of the two orderings in all cases, without any explicit information about the operators' relative costs.

The eddy's effectiveness in this scenario is due to simple fluid dynamics, arising from the different rates of consumption by $s1$ and $s2$. Recall that the edges in a River dataflow graph correspond to fixed-size queues. This limitation has the same effect as *back-pressure* in a fluid flow: production along the input to any edge is limited by the rate of consumption at the output. The lower-cost selection (e.g., $s1$ at the left of Figure 4) can consume tuples more quickly, since it spends less time per tuple; as a result the lower-cost operator exerts less back-pressure on the input table. At the same time, the high-cost operator *produces* tuples relatively slowly, so the low-cost operator will rarely be required to consume a high-priority, previously-seen tuple. Thus most tuples are routed to the low-cost operator first, even though the costs are not explicitly exposed or tracked in any way.

4.3 Fast Eddy: Learning Selectivities

This naive eddy works well for handling operators with different costs but equal selectivity. But we have not yet considered differences in selectivity. In our second experiment we keep the costs of the operators constant and equal (5 units), keep the selectivity of $s2$ fixed at 50%, and vary the selectivity of $s1$ across runs. The results in Figure 5 are less encouraging, showing eddy performing as we originally expected, about half-way between the best and worst plans. Clearly our naive priority scheme and the resulting back-pressure are insufficient to capture differences in selectivity.

To resolve this dilemma, we would like our priority scheme to favor operators based on both their consumption and production rate. Note that the consumption (input) rate of an operator is determined by cost alone, while the production (output) rate is determined by a product of cost and selectivity. Since an operator's back-pressure on its input depends largely on its consumption rate, it is not surprising that our naive scheme does not capture differing selectivities.

In order to track both consumption and production over time, we enhance our priority scheme with a simple learning algorithm based on *Lottery Scheduling* [WW94]. Each time the eddy gives a tuple to an operator, it credits the operator one "ticket". Each time the operator returns a tuple to the eddy, one ticket is debited from the eddy's running count for that operator. When an eddy is ready to send a tuple to be processed, it "holds a lottery" among the operators eligible for receiving the tuple. (The interested reader is

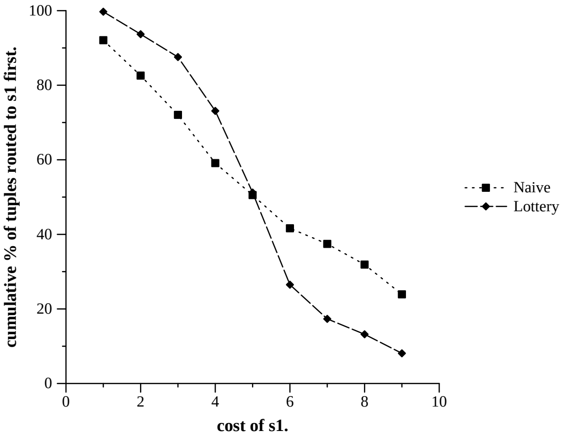

Figure 8: Tuple flow with lottery scheme for the variable-cost experiment(Figure 6).

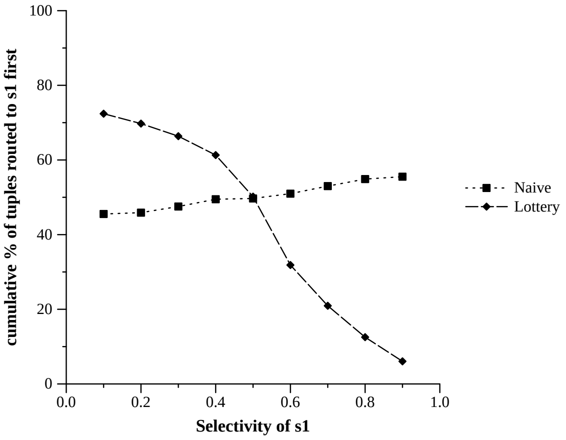

Figure 9: Tuple flow with lottery scheme for the variable-selectivity experiment(Figure 7).

referred to [WW94] for a simple and efficient implementation of lottery scheduling.) An operator's chance of "winning the lottery" and receiving the tuple corresponds to the count of tickets for that operator, which in turn tracks the relative efficiency of the operator at draining tuples from the system. By routing tuples using this lottery scheme, the eddy tracks ("learns") an ordering of the operators that gives good overall efficiency.

Figures 6 and 7 repeat our previous experiments, this time measuring the more intelligent lottery-based routing scheme against the naive back-pressure scheme, along with the two static orderings. The lottery-based scheme handles both scenarios quite effectively, slightly improving the eddy in the changing-cost experiment, and performing almost optimally in the changing-selectivity experiment.

To explain this a bit further, in Figures 8 and 9 we display the percent of tuples that followed the order $s1, s2$ (as opposed to $s2, s1$) in the two eddy schemes; this roughly represents the average ratio of lottery tickets possessed by $s1$ and $s2$ over time. Note that the naive back-pressure policy is barely sensitive to changes in selectivity, and in fact drifts slightly in the wrong direction as the selectivity of $s1$ is increased. By contrast, the lottery based scheme shows benefits in both experiments, even improving the back-pressure scheme in the changing-cost experiment.

In both graphs one can see that when the costs and selectivities are close to equal, then the percentage of tuples that go the right way is close to 50%. This latter observation is intuitive, but actually quite significant. The lottery-based eddy approaches the *cost* of an optimal ordering, but does not concern itself about strictly observing the optimal ordering. This is in contrast to earlier papers on dynamic reoptimization [KD98, UFA98, IFF+99], which explicitly run a traditional query optimizer during processing to determine the optimal plan at a given time. By focusing on overall cost rather than on finding the optimal plan, the lottery scheme probabilistically provides nearly optimal performance with much less effort, allowing re-optimization to be done with an extremely lightweight technique that can be executed multiple times for every tuple.

A related observation is that the lottery algorithm gets closer to perfect routing ($y = 0\%$) on the right of Figure 9 than it does ($y = 100\%$) on the left. Yet in the corresponding performance graph (Figure 7), the differences between the lottery-based eddy and the optimal static ordering do not change much across selectivity settings. This phenomenon is explained by examining the "jeopardy" of making ordering errors in either case. Consider the left side of the graph, where the selectivity of $s1$ is 10%, $s2$ is 50%, and the costs of each are $c = 5$ delay units. Let e be the rate at which tuples are routed erroneously (to $s2$ before $s1$ in this case). Then the expected cost of the query is $(1-e) \cdot 1.1c + e \cdot 1.5c = .4ec + 1.1c$. By contrast, in the second case where the selectivity of $s1$ is changed to 90%, the expected cost is $(1-e) \cdot 1.5c + e \cdot 1.9c = .4ec + 1.5c$. Since the jeopardy is higher at 90% than at 10%, the lottery more aggressively favors the optimal ordering at 90% selectivity than at 10%.

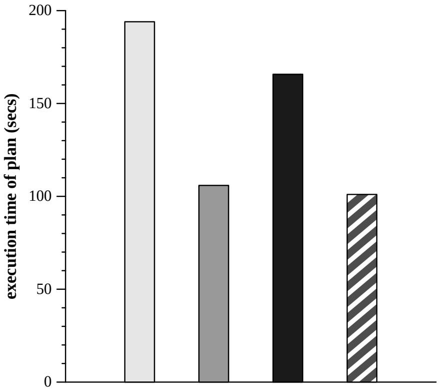

Figure 10: Performance of two joins: a selective Index Join and a Hash Join

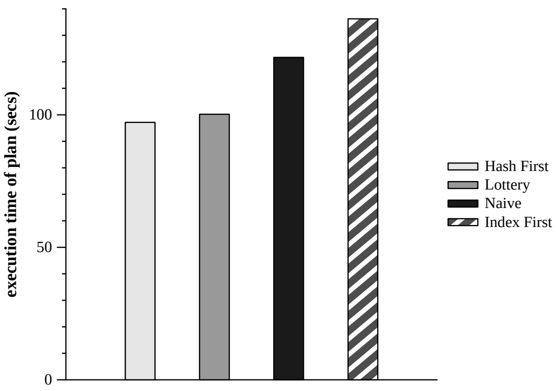

Figure 11: Performance of two joins: a less selective Index Join and a Hash Join

4.4 Joins

We have discussed selections up to this point for ease of exposition, but of course joins are the more common expensive operator in query processing. In this section we study how eddies interact with the pipelining ripple join algorithms. For the moment, we continue to study a static performance environment, validating the ability of eddies to do well even in scenarios where static techniques are most effective.

We begin with a simple 3-table query:

SELECT	*
FROM	R, S, T
WHERE	$R.a = S.a$
AND	$S.b = T.b$

In our experiment, we constructed a preoptimized plan with a hash ripple join between R and S, and an index join between S and T. Since our data is uniformly distributed, Table 2 indicates that the selectivity of the RS join is 1.8×10^{-4}; its selectivity *with respect to* S is 180% – i.e., each S tuple entering the join finds 1.8 matching R tuples on average [Hel98]. We artificially set the selectivity of the index join w.r.t. S to be 10% (overall selectivity 1×10^{-5}). Figure 10 shows the relative performance of our two eddy schemes and the two static join orderings. The results echo our results for selections, showing the lottery-based eddy performing nearly optimally, and the naive eddy performing in between the best and worst static plans. To confirm the flexibility of the lottery scheme, we change the selectivity of the index join w.r.t. S to be 90%. The resulting performance in Figure 11 shows that a different static plan is optimal, and the lottery-based eddy still performs competitively.

As noted in Section 2.4.2, index joins are very analogous to selections. Hash joins have more complicated and symmetric behavior, and hence merit additional study. Figure 12 presents performance of two versions of this query. We change the data in R, S and T so that the selectivity of the ST join w.r.t. S is 20% in one version, and 180% in the other. In all runs, the selectivity of the RS join predicate w.r.t. S is fixed at 100%. As the figure shows, the lottery-based eddy continues to perform nearly optimally.

Figure 13 shows the percent of tuples that follow one order or the other in all four join experiments. While the eddy is not strict about following the optimal ordering, it is quite close in the case of the experiment where the hash join should precede the index join. In this case, the relative cost of index join is so high that the jeopardy of choosing it first drives the hash join to nearly always win the lottery.

4.5 Responding to Dynamic Fluctuations

Eddies should adaptively react over time to the changes in performance and data characteristics described in Section 1.1. The routing schemes described up to this point have not considered how to achieve this. In

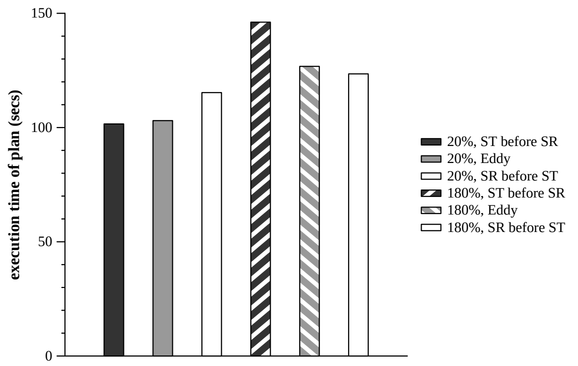

Figure 12: Performance of hash joins $R \bowtie S$ and $S \bowtie T$. $R \bowtie S$ has selectivity 100% w.r.t. S, the selectivity of $S \bowtie T$ w.r.t. S varies between 20% and 180% in the two runs.

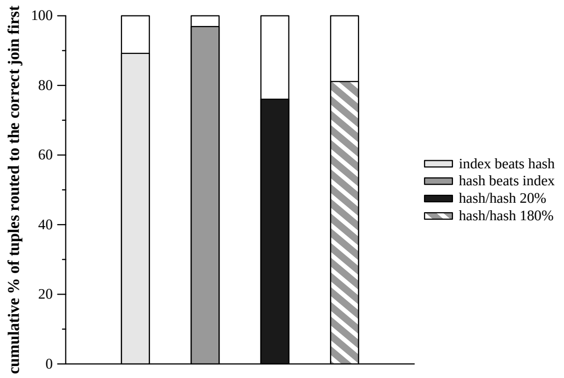

Figure 13: Percent of tuples routed in the optimal order in all of the join experiments.

particular, our lottery scheme weighs all experiences equally: observations from the distant past affect the lottery as much as recent observations. As a result, an operator that earns many tickets early in a query may become so wealthy that it will take a great deal of time for it to lose ground to the top achievers in recent history.

To avoid this, we need to modify our point scheme to forget history to some extent. One simple way to do this is to use a *window* scheme. In this scheme, time is partitioned into windows, and the eddy keeps track of two counts for each operator: a number of *banked* tickets, and a number of *escrow* tickets. Banked tickets are used when running a lottery. Escrow tickets are used to measure efficiency during the window. At the beginning of the window, the value of the escrow account replaces the value of the banked account (i.e., `banked = escrow`), and the escrow account is reset (`escrow = 0`). This scheme ensures that operators "re-prove themselves" each window. The window length needs to be set carefully to balance the system's reaction time against its sensitivity to transient fluctuations; we return to this point in Section 6.

We consider a scenario of a three-table join, where two of the tables are external and used as "inner" relations by index joins. Our third relation has 4,000 tuples. This scenario occurs when joining a set of data to remote information resources that can do lookups based on input values (variable bindings); e.g., a directory server, GIS server, or other form-based search interface. Since we assume that the server is remote, we implement the "cost" in our index module as a time delay (i.e., `while (gettimeofday() < x) ;`) rather than a spin loop; this better models the behavior of waiting on an external event like a network response. In this experiment we always have one index that is fast (1 time unit per lookup) and another that is slow (100 time units per lookup), but we switch speeds on the two indexes every 1000 calls. That is, three times during the scan of S the fast index becomes slow, and the slow index becomes fast. Both indexes return a single matching tuple 10% of the time.

Figure 14 shows the performance of both static plans, compared with an eddy using a lottery with a window scheme. As we would hope, the eddy is much faster than either static plan. Figure 15 shows that the eddy adapts correctly to changes, switching orders when the operator costs switch. The eddy sends most of the first 1000 tuples to index #1 first, which starts off cheap, as reflected by the growth to a 100% value in the first quarter of the graph. It sends most of the second 1000 tuples to index #2 first, causing the overall percentage of tuples to reach about 50% in the graph, as reflected by the near-linear drift toward 50% in the second quarter of the graph. This pattern repeats in the third and fourth quarters, with the eddy eventually displaying an even use of the two orderings over time – always favoring the best ordering.

Each static plan is suboptimal for two out of four runs of 1000 tuples, and the ratio of a suboptimal plan to an optimal plan is $(100 + 0.1 \cdot 1)/(1 + 0.1 \cdot 100) = 9.1$. Thus the ratio of a static execution to a perfect dynamic optimizer would be $(2 \cdot 9.1 + 2 \cdot 1)/(4 \cdot 1) = 5.05$. Figure 14 shows that the eddy does not

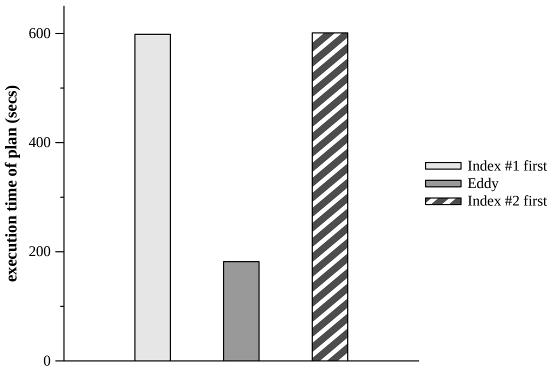

Figure 14: Adapting to changing join costs: performance.

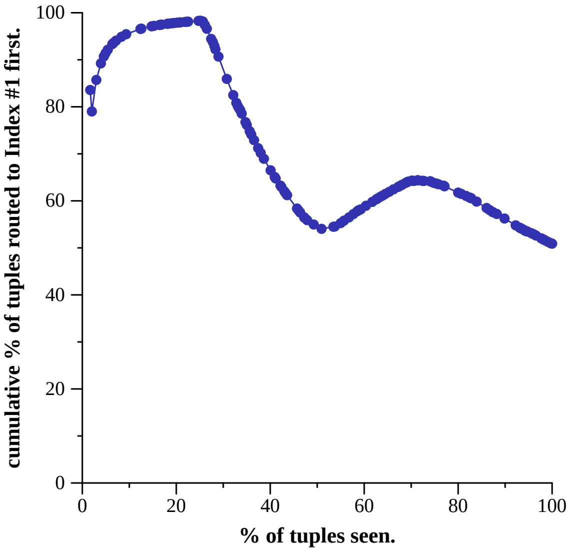

Figure 15: Adapting to changing join costs: tuple movement.

do perfectly, achieving only about a 3.3x speedup; we believe this has to do with our window scheme, and we intend to study this more carefully in future work.

For brevity, we omit here a similar experiment in which we modified selectivity over time. The results were similar, except that changing only the selectivity of two operators results in less dramatic benefits for an adaptive scheme. This can be seen analytically. The ratio between a static and optimal dynamic ordering is maximized when selectivities vacillate evenly between 100% and 0%, say $2k$ times with any cost c. The resulting ratio of static to optimal-dynamic is $(k \cdot 2c + k \cdot 1c)/(2k \cdot 1c) = 3/2$. With more operators, adaptivity to changes in selectivity can become more significant, however.

4.5.1 Delayed Delivery

As a final experiment, we study the case where an input relation suffers from an initial delay, as in [AFTU96, UFA98]. We return to the 3-table query shown in the left of Figure 12, with the RS selectivity at 100%, and the ST selectivity at 20%. We delay the delivery of R by 10 seconds; the results are shown in Figure 16. Unfortunately, we see here that our eddy – even with a lottery and a window-based forgetting scheme – does not adapt to initial delays of R as well as it could. Figure 17 tells some of the story: in the early part of processing, the eddy incorrectly favors the RS join, even though no R tuples are streaming in, and even though the RS join should appear second in a normal execution (Figure 12). The eddy does this because it observes that the RS join does not produce any output tuples when given S tuples. So the eddy awards most S tuples to the RS join initially, which places them in an internal hash table to be subsequently joined with R tuples when they arrive. The ST join is left to fetch and hash T tuples. This wastes resources that could have been spent joining S tuples with T tuples during the delay, and "primes" the RS join to produce a large number of tuples once the Rs begin appearing.

Note that eddy does better than pessimally: when R begins producing tuples (at 43.5 on the x axis of Figure 17), the S values bottled up in the RS join burst forth, and the eddy quickly throttles the RS join, allowing the ST join to process most tuples first. This scenario indicates two problems with our implementation. First, our ticket scheme does not capture the growing selectivity inherent in a join with a delayed input. Second, storing tuples inside the hash tables of a single join unnecessarily prevents other joins from processing them; it might be conceivable to hash input tuples within multiple joins, if care were

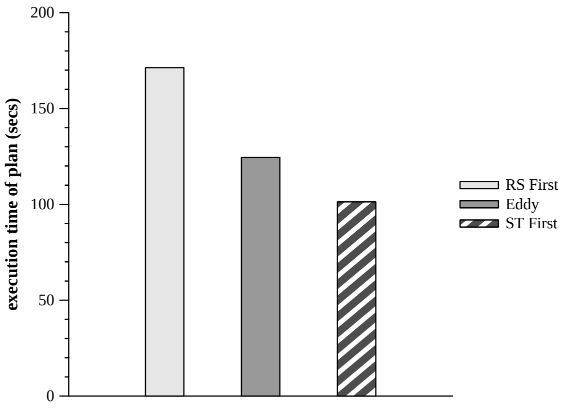

Figure 16: Adapting to an initial delay on R: performance

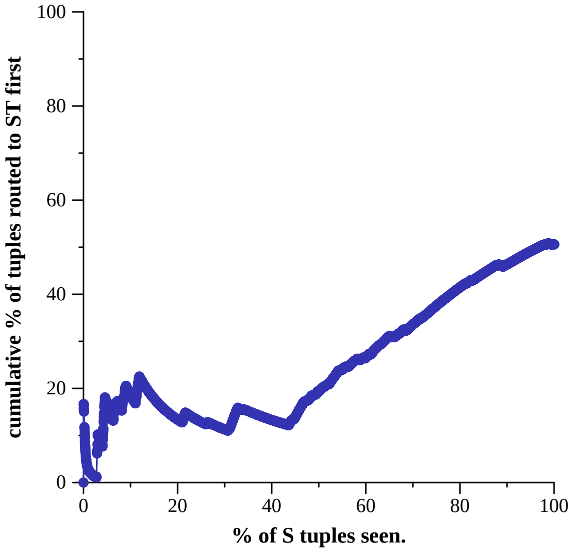

Figure 17: Adapting to an initial delay on R: tuple movement.

taken to prevent duplicate results from being generated. A solution to the second problem might obviate the need to solve the first; we intend to explore these issues further in future work.

For brevity, we omit here a variation of this experiment, in which we delayed the delivery of S by 10 seconds instead of R. In this case, the delay of S affects both joins identically, and simply slows down the completion time of all plans by about 10 seconds.

5 Related Work

As noted in the introduction, there has been a host of both research and industrial work on logically unifying information sources into a single system view, and attempting to ease the difficulty of data integration. This is important groundwork for systems like Telegraph, but is orthogonal to the discussion of this paper

[IFF+99] observantly notes that the original INGRES query decomposition scheme was integrated with the one-variable query processor, and in essence did run-time optimization [SWK76]. However, the ordering heuristic in INGRES was static, and could have been applied before the query execution began.

There has been a variety of work on the broad topic of generating more than one possible plan for a query. A number of papers treat the question of compiling parameterized plans that are instantiated at run-time [INSS97, GC94], but these schemes do not change the plan once it is in flight, and hence do not address the problems of our environment. DEC Rdb (subsequently Oracle Rdb) was quite early in considering competitive techniques for choosing among different access methods at run-time [AZ96]. The goal in that work was to actually measure costs of alternative access methods at run-time rather than resort to estimation, but after a short measurement phase the chosen access method was assumed to be correct for the rest of the query execution. This bears some resemblence to the work on sampling for cost estimation (see [BDF+97] for a survey).

The initial work on Query Scrambling [AFTU96] studied the problem of network unpredictabilities in processing queries over wide-area sources. One important technique they introduced was to materialize data from remote sources even when query processing was blocked waiting for other sources. This idea can be used to prefetch data from an input even when the operator has blocked it waiting for a barrier; this can be used in concert with the ideas of this paper. This materialization can ameliorate the problem of barriers, but does not solve it; the local work postponed until after a barrier can still be quite significant. Later results from this team focus on rescheduling runnable sub-plans in the face of initial delays in delivery, with an eye toward using spare cycles during delays to pay for materialization and subsequent re-scans [UFA98]. In focusing on initial delays, this work does not attempt to reorder in-flight operators as we do here. However, it does introduce the suggestive notion of *operator synthesis*, i.e., changing the set of operators in a query plan during processing. We intend to explore this idea further in the eddy framework, combining it with the ability to reorder pipelines.

To our knowledge, this paper represents the first general query processing scheme for reordering in-flight operators within a pipeline. Ng, et al. suggest allow reordering operators in parallel dataflow pipelines, but only consider unary operators [NWMN99], not the more common (and more difficult) case of joins.

There is a recently growing body of work on reoptimizing operators at the completion of pipelines, including work by Kabra and DeWitt [KD98], and the Tukwila system [IFF+99]. These schemes reorder operators only after materializing temporary results, i.e. after blocks in the query plan. They contain explicit (and sometimes programmable) logic for discretely deciding when to perform reoptimization (as does the scheme of [NWMN99]. These inter-pipeline techniques are not adaptive in the sense used in traditional control theory (e.g., [Son98]) or (more recently) machine learning (e.g., [Mit97]); they make decisions without any feedback from the operations they are to optimize, instead performing static optimizations at coarse-grained intervals in the query plan. It is important to note that these efforts are complementary to the work here: eddies can be used to do tuple scheduling within pipelines, and techniques like those of [UFA98, KD98, IFF+99] can be used to reoptimize across pipelines. Of course such a marriage sacrifices the simplicity of eddies, requiring both the traditional complexity of cost estimation and plan enumeration along with the ideas of this paper. There are also significant questions on how best to combine these techniques – e.g., how many materialization operators to put in a plan, which operators to put in which eddy pipelines, etc.

Both the Tukwila and Query Scrambling teams saw benefits in the pipelined hash join, and each proposed an efficient out-of-core version [IFF+99, UF99]. In both cases, the motivation for using pipelined hash join was its lack of barriers, not the frequent symmetry allowing for reordering. The X-Join [UF99] enhances the pipelined hash join both by running efficiently in the out-of-core case, and by exploiting delay time to aggressively match previously-received (and spilled) tuples. We intend to experiment with X-Joins and eddies; X-Joins change cost when they begin to spill tuples to disk, and hence are best handled by adaptive optimization.

Our characterization of barriers and moments of symmetry appears to be new, arising as it does from our interest in reoptimizing pipelines. Traditional work on parallel query processing recognized the power of pipelining for parallelism [DG92]. Wilschut and Apers' work on pipelining hash joins was motivated both by the opportunity for pipelined parallelism, and by the lack of barriers in the algorithm that enabled inputs to run at asynchronous rates. [IFF+99] incorrectly argues that index join is not pipelining; pre-indexed collections are widely available in database systems and in network services. [NWMN99] warns of the complications of reordering operators at arbitrary points in processing, but does not analyze any specific algorithms in this regard.

The Control project [HAC+99] addresses the goal of making long-running computations interactive, using techniques like Online Aggregation to return refining approximate results during query processing, and allowing for in-flight user control of the processing. This includes work on ripple joins [HH99], which are streaming join algorithms that adapt to statistical characteristics of data. There is a natural synergy between the goals of the Control project and the development of large-scale adaptive query processors; techniques used to pipeline best-effort answers in online query processing are naturally adaptive to less-than-perfect performance scenarios. The need for optimizing pipelines in the Control project initially motivated our work on eddies [6].

The River project [ADAT+99] was another main inspiration of this work. River arose from frustrations in achieving a sorting record on a cluster of workstations with NowSort [ADADC+97]: impressive performance was possible in the best case, but even slight perturbations in the cluster (e.g., a stray file occupying the outer tracks of a scratch disk) would cause significant slowdowns in the common case. River uses two

[6] The Control project [HAC+99] is not explicitly related to the field of control theory [Son98], though Eddy appears to link the two in some regards.

basic techniques to attack performance heterogeneity and unpredictability for I/O operations in clusters of computers: work is balanced among consumers via a highly optimized distributed queue implementation, while work is balanced among producers via a data redundancy mechanism. River allows modules to work as fast as they can, naturally balancing the flow to whichever modules are faster. We carried the River philosophy into the intial back-pressure design of eddies, and intend to return to the parallel load-balancing aspects of the optimization problem in future work.

6 Conclusions and Future Work

An eddy is a query processing mechanism that encapsulates a set of pipelined operators, and adaptively routes tuples through them. Eddies are particularly beneficial in the unpredictable query processing environments prevalent in wide-area systems. They fit naturally with join algorithms from the Ripple Join family, which have frequent moments of symmetry and short or non-existent barriers. Eddies can be used as the sole optimization mechanism in a query processing system, obviating the need for much of the complex code required in a traditional query optimizer. Alternatively, eddies can be used in concert with traditional optimizers to improve adaptability within pipelines. Our initial results indicate that eddies perform well under a variety of circumstances, though some questions remain in improving reaction time and in adaptively choosing join orders with delayed sources. We are suficiently encouraged by these early results that we are using eddies and River as the basis for query processing in the Telegraph Global DBMS.

In order to focus our energies in this initial work, we have explicitly postponed a number of questions in understanding, tuning, and extending these results. Foremost among the remaining challenges is to formally prove that eddies converge quickly to a near-optimal execution in static scenarios, and that they adaptively converge when conditions change – both for selections and for joins, including hash joins that "absorb" tuples into their hash tables. We intend to focus on multiple performance metrics, including time to completion, the rate of output from a plan, and the rate of refinement for online aggregation estimators. We are also interested in analyzing the ability of eddies to effectively order dependent predicates [NGMC98]. Our formal research agenda is likely to involve more careful analysis of lotteries and forgetting schemes, perhaps based on techniques borrowed from control theory or machine learning. In a related vein, we would like to automatically tune the aggressiveness with which we forget past observations, so that we avoid introducing a tuning knob to adjust window-length or some analogous constant (e.g., a hysteresis factor).

Another main goal is to attack the remaining static aspects of our scheme: the pre-optimization choices of spanning tree and join algorithms. We believe that competition is key here: one can run multiple redundant joins and join algorithms and track their behavior in an eddy, adaptively choosing among them over time. The implementation challenge in that scenario relates to preventing duplicates from being generated, while the efficiency challenge comes in not wasting too many computing resources on unpromising alternatives. We suspect that tagging techniques analogous to those of [IFF+99, UF99] can be applied for duplicate elimination in this framework, aided perhaps by Bloom filters to reduce the need for duplicate checks when possible, and query-scrambling-like postponement of any post-Bloom-filter work until delays appear or the query must be completed. With regard to efficiency, we are hopeful that the basic mechanism of eddy will moderate the resources spent on inefficient alternative operators.

A third major challenge is to harness the parallelism and adaptivity available to us in River. We concur with [Bar99, Gra99] that massively parallel systems are reaching their limit of manageability, even as data sizes continue to grow very quickly. Adaptive techniques like eddies and rivers can significantly aid in the manageability of a new generation of massively parallel query processors. Rivers have been shown to adapt gracefully to performance changes in large clusters, spreading query processing load across nodes and spreading data delivery across data sources. Eddy faces additional challenges to meet the promise of rivers: in particular, reoptimizing queries with intra-operator parallelism entails repartitioning data, which adds an expense to reordering that was not present in our single-site eddies. An additional complication arises when trying to adaptively adjust the degree of partitioning for each operator in a plan. On a similar note, we would like to explore enhancing eddies and rivers to tolerate failures of sources or of participants in parallel execution.

Finally, we are exploring the application of eddies and rivers to the generic space of dataflow programming, including applications such as multimedia analysis and transcoding, and the composition of scalable, reliable internet services [FGCB97, GWBC99]. Our intent is for rivers to be a generic parallel dataflow engine, and for eddies to be the main scheduling mechanism in that environment.

Acknowledgments

Vijayshankar Raman provided significant assitance in the course of this work. Remzi Arpaci-Dusseau, Eric Anderson and Noah Treuhaft implemented River, and helped get Eddy up and running. Mike Franklin asked a variety of hard questions, and suggested interesting directions for future work. Stuart Russell, Christos Papadimitriou, Alistair Sinclair, Kris Hildrum and Lakshminarayanan Subramanian all helped us focus our thoughts on the formal aspects of eddies. Thanks to both Navin Kabra and Mitch Cherniack for initial suggestive discussions on run-time reoptimization, and to the database group at Berkeley for feedback. Stuart Russell suggested the term "eddy".

This work was supported by a grant from IBM Corporation, NSF grant IIS-9802051, and a Sloan Foundation Fellowship. Computing and network resources for this research were provided through NSF RI grant CDA-9401156.

References

[ACPS96] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Montreal, 1996.

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Tucson, May 1997.

[ADAT+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, and Katherine Yelick. Cluster I/O with River: Making the Fast Case Common. In *Sixth Workshop on I/O in Parallel and Distributed Systems (IOPADS '99)*, pages 10–22, Atlanta, May 1999.

[AFTU96] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling Query Plans to Cope With Unexpected Delays. In *4th International Conference on Parallel and Distributed Information Systems (PDIS)*, Miami Beach, December 1996.

[Aok99] Paul M. Aoki. How to Avoid Building DataBlades That Know the Value of Everything and the Cost of Nothing. In *11th International Conference on Scientific and Statistical Database Management*, Cleveland, July 1999.

[AZ96] Gennady Antoshenkov and Mohamed Ziauddin. Query Processing and Optimization in Oracle Rdb. *VLDB Journal*, 5(4):229–237, 1996.

[Bar99] Robert Barnes. Scale Out. In *High Performance Transaction Processing Workshop (HPTS '99)*, Asilomar, September 1999. http://www.research.microsoft.com/barc/hpts99.

[BDF+97] Daniel Barbara, William DuMouchel, Christos Faloutsos, Peter J. Haas, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Theodore Johnson, Raymond T. Ng, Viswanath Poosala, Kenneth A. Ross, and Kenneth C. Sevcik. The New Jersey Data Reduction Report. *IEEE Data Engineering Bulletin*, 20(4), December 1997.

[BO99] J. Boulos and K. Ono. Cost Estimation of User-Defined Methods in Object-Relational Database Systems. *SIGMOD Record*, 28(3):22–28, September 1999.

[BVO97] J. Boulos, Y. Viémont, and K. Ono. Analytical Models and Neural Networks for Query Cost Evaluation. In *Proc. 3rd International Workshop on Next Generation Information Technology Systems*, Neve Ilan, Israel, 1997.

[DG92] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, 1992.

[DGS+90] David J. DeWitt, Shahram Ghandeharizadeh, Donovan 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, Mar 1990.

[DKO+84] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 1–8, Boston, June 1984.

[FGCB97] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Cluster-Based Scalable Network Services. In *Proc. 1997 Symposium on Operating Systems Principles (SOSP-16)*, St-Malo, France, October 1997.

[FMLS99] Daniela Florescu, Ioana Manolescu, Alon Levy, and Dan Suciu. Query Optimization in the Presence of Limited Access Patterns. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Phildelphia, June 1999.

[GC94] G. Graefe and R. Cole. Optimization of Dynamic Query Evaluation Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, 1994.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[Gra90] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 102–111, Atlantic City, May 1990.

[Gra99] Jim Gray. How High is High-Performance Transaction Processing? In *High Performance Transaction Processing Workshop (HPTS '99)*, Asilomar, September 1999. http://www.research.microsoft.com/barc/hpts99.

[GWBC99] Steven D. Gribble, Matt Welsh, Eric A. Brewer, and David Culler. The MultiSpace: an Evolutionary Platform for Infrastructural Services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, June 1999.

[HAC+99] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. Interactive Data Analysis: The Control Project. *IEEE Computer*, 32(8):51–59, August 1999.

[HBF99] Joseph M. Hellerstein, Eric Brewer, and Michael Franklin. A Storage Manager for Telegraph. In *High Performance Transaction Processing Workshop (HPTS '99)*, Asilomar, September 1999. http://www.research.microsoft.com/barc/hpts99.

[Hel98] Joseph M. Hellerstein. Optimization Techniques for Queries with Expensive Methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 287–298, Philadelphia, 1999.

[HKWY97] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proc. 23rd International Conference on Very Large Data Bases (VLDB)*, Athens, 1997.

[HSC99] Joseph M. Hellerstein, Michael Stonebraker, and Rick Caccia. Open, Independent Enterprise Data Integration. *IEEE Data Engineering Bulletin*, 22(1), March 1999. http://www.cohera.com.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc Fiedman, Alon Levy, and Daniel S. Weld. An Adaptive Query Execution System for Data Integration. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Philadelphia, 1999.

[IK84] Toshihide Ibaraki and Tiko Kameda. Optimal Nesting for Computing N-relational Joins. *ACM Transactions on Database Systems*, 9(3):482–502, October 1984.

[INSS97] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric Query Optimization. *VLDB Journal*, 6(2):132–151, 1997.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *Proc. 12th International Conference on Very Large Databases (VLDB)*, pages 128–137, August 1986.

[KD98] Navin Kabra and David J. DeWitt. Efficient Mid-Query Reoptimization of Sub-Optimal Query Execution Plans. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 106–117, Seattle, 1998.

[Met97] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the Usenix 1997 Technical Conference*, Anaheim, January 1997.

[Mit97] Tom Mitchell. *Machine Learning*. McGraw Hill, 1997.

[NGMC98] Shivakumar Narayana, Hector Garcia-Molina, and Chandra S. Chekuri. Filtering with Approximate Predicates. In *Proceedings of 1998 International Conference on Very Large Databases (VLDB'98)*, New York, August 1998.

[NWMN99] Kenneth W. Ng, Zhenghao Wang, Richard R. Muntz, and Silvia Nittel. Dynamic Query Re-Optimization. In *11th International Conference on Scientific and Statistical Database Management*, Cleveland, July 1999.

[RPK+99] B. Reinwald, H. Pirahesh, G. Krishnamoorthy, G. Lapis, B. Tran, and S. Vora. Heterogeneous Query Processing Through SQL Table Functions. In *15th International Conference on Data Engineering*, pages 366–373, Sydney, March 1999.

[RRH99] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. In *Proc. 25th International Conference on Very Large Data Bases (VLDB)*, pages 709–720, Edinburgh, 1999.

[SBH98] M. Stonebraker, P. Brown, and M. Herbach. Interoperability, Distributed Applications, and Distributed Databases: The Virtual Table Interface. *IEEE Data Engineering Bulletin*, 21(3):25–34, September 1998.

[Sha86] Leonard D. Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[Son98] Eduardo D. Sontag. *Mathematical Control Theory: Deterministic Finite-Dimensional Systems, Second Edition.* Number 6 in Texts in Applied Mathematics. Springer-Verlag, New York, 1998.

[SWK76] M. R. Stonebraker, E. Wong, and P. Kreps. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[Tho99] Anne Thomas. webMethods B2B Integration Server: Business-to-Business E-commerce Solutions. Technical report, Patricia Seybold Group, 1999. Available from http://www.webmethods.com/products/whitepapers/index.html.

[UF99] Tolga Urhan and Michael Franklin. XJoin: Getting Fast Answers From Slow and Bursty Networks. Technical Report CS-TR-3994, University of Maryland, February 1999.

[UFA98] Tolga Urhan, Michael Franklin, and Laurent Amsaleg. Cost-Based Query Scrambling for Initial Delays. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Seattle, June 1998.

[vECG+92] T. von Eicken, D. E. Culler, S. C. Goldstein, , and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th ISCA*, pages 256–266, May 1992.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proc. First International Conference on Parallel and Distributed Info. Sys. (PDIS)*, pages 68–77, 1991.

[WW94] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proc. of the First Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 1–11, Monterey, CA, November 1994. USENIX Assoc.

A The Costs of Adaptivity

As observed previously, we favor adaptivity over raw performance in the best case. It is important to note that adaptivity does come at a cost in resource utilization.

Pipelining always incurs a memory penalty. Even with traditional join algorithms like hybrid hash joins, a large pipeline of joins has a large memory footprint: for a query on relations $R_1, ..., R_k$, at least $(k-1)\sqrt{\min_{i=1}^{k} |R_i|}$ memory will be used, or recursive partitioning will be required [Sha86].

Within a single join, pipelining often comes at the expense of additional resource consumption as well. While hybrid hash join has memory requirements equal to $\sqrt{\min(|R_1|, |R_2|)}$, pipelined hash joins like hash ripple require memory equal to $\sqrt{|R_1|} + \sqrt{|R_2|}$. When memory becomes scarce, the space consumption must be converted to time via recursive partitioning.

One is not required to pipeline entire query plans, and it is always possible to choose blocking operators like hybrid hash join. In such scenarios, the reordering may be constrained. For example, if materialization operators are introduced into the tree, joins below the materialization operator cannot be reordered on the fly to a point above the materialization.

In our initial work, we assume in Telegraph that memory and disk I/O can be spent in exchange for aggressive pipelining. Part of our reasoning is to maximally exercise our reordering ideas; another idea is to preserve the opportunity for online aggregation and other "Control" interactions [HAC+99], which depend on non-blocking operators. We continue to entertain the idea of merging eddies with traditional static optimization in Telegraph in various ways, but are postponing that decision until we are convinced that static optimization will solve significant problems that eddies cannot address.