The Design of Telegraph: Adaptive Dataflow for Streams

Authors
Computer Science Division
UC Berkeley
telegraph@db.cs.berkeley.edu

Abstract
Numerous emerging applications exploit a combination of data-intensive services and complex networking fabrics; prominent examples include ubiquitous sensor networks, peer-to-peer systems, continuous queries, and deep web querying. These applications share a common architectural theme: the composition of operators over unpredictable streams of networked data. In this paper, we present Telegraph, an adaptive dataflow system for managing streams of networked data. Telegraph provides an extensible dataflow architecture appropriate both for actively streaming data, and for “still” data that can be streamed from storage servers into the system. Telegraph provides adaptive components for flow optimization, fault tolerance, and resource sharing; it encapsulates these components as operators in a flow, thus masking the complexity of these features from authors of dataflows or new dataflow operators.

In the paper we describe our motivation for building an adaptive dataflow infrastructure, and highlight the need for a mix of traditional database system design goals with those of networking systems. We present the design of the first version of Telegraph, report on early applications we have built, and present examples of performance benefits to justify our architectural decisions. We also highlight a number of new research challenges in systems infrastructure for unpredictable dataflows, and discuss the relationship between system infrastructure for streams and a variety of application semantics.

1 Context and Overview

A number of emerging applications exploit a combination of data-intensive services and global networking fabrics. The evolution of the "deep web" is one clear example [?], in which centralized databases and data feeds are being made available via the Internet, with a variety of value-added services like comparison shopping or supply chain management stitching such centralized services together into more global federations [?]. This phenomenon is not limited to the web, however. Peer-to-peer filesharing services like Napster and Gnutella represent a new "channel" for data movement on the Internet, which has different administrative, networking, and interface characteristics than the web. Perhaps more revolutionary is the movement towards networks of ubiquitous sensors and actuators [?], with numerous potentially world-changing applications in seismic monitoring, biological fieldwork, medical monitoring and drug delivery, and fine-grained analogs to current supply chain, retail and marketing applications.

In all of these data-intensive networked applications, the architectural focus is necessarily around streams of data that flow through and are processed by components of the system. In many emerging applications, these data streams pour continuously out of data feeds such as sensors and news wires. In other cases, stationary data is made to flow, say by initiating or sharing a scan from storage (e.g., [?]), or by pushing data onto a communications channel (e.g. [?, ?]). Given the centrality of streaming data in these environments, a natural challenge is the development of a retargetable infrastructure of dataflow software to support these applications.

Of course, dataflow has long been a fundamental concept in database systems. As far back as the initial Ingres and System R designs, relational database query processors have been built around an internal API of composable iterators that route tuples from disk-based access methods through a reusable library of other operators including sorting, selection, and join algorithms. The distributed implementations of these systems adjusted the basic dataflow model in order to more efficiently use network resources [?, ?, ?]. Dataflow played an even more prominent role in the development of Shared Nothing parallel database systems [?, ?, ?] where streams of tuples must be explicitly partitioned across query operators that are replicated on the nodes of a cluster of machines.
Despite the importance of the dataflow architecture in these traditional database architectures, it has usually played a less central role than the driving concerns about processing. That is, systems have been designed around relational query processing operators, and dataflow has been considered mainly as "plumbing" in service of those operators. As we move from a world of isolated machines or tightly-coupled data processing centers into a world of dynamic, loose federations of widely disparate data sources and services, dataflow becomes a primary concern. In particular, the highly dynamic, multi-party, hazard-ridden nature of this new environment requires a flexible and adaptive approach to dataflow in data-intensive systems.

1.1 Database and Networking Convergence

For inspiration on how to design a robust system that can manage dataflow in a large and inhospitable environment, it is useful to look at the Internet itself. The Internet is composed of machines and software that route streams of packets both physically across networks, and also through code modules like packet filters, protocol layers, multicast trees, and other dataflow operations. Query processing and network routing systems are, if anything, becoming more similar over time: recent designs in modular routers and operating systems for network appliances look remarkably like extensible relational query processing engines. It is intriguing and perhaps suggestive that the similarities between these systems have not been highlighted more explicitly in the literature.

One important distinction is that these systems were designed with very different engineering constraints and goals. Relational engines were designed in a top-down fashion: given a declarative query language and a mapping to a physical algebra, the challenge was to write a self-contained system that could optimize performance by exploiting data and query semantics, data and machine statistics, local storage, and implementation of particular physical operators. Because these systems had declarative interfaces and devoted administrators, arbitrary complexity (decades of evolutionary research and development on the same software engines!) could be hidden behind those interfaces if they made the system run faster. This top-down approach was supported by deploying the systems in situations where there was certain amount of control over the environment. For example, inherent in these architectures is the notion of a Data Base Administrator (DBA) and a support staff who have the ability to closely monitor and adjust the properties of the system.

In contrast, networking infrastructures were designed under a much different set of assumptions. In particular, they were designed to graciously cope with a constantly changing, growing, and fundamentally uncontrollable and unreliable set of components. For example, a fundamental assumption underlying the Internet Protocol is that the system will never be 100% operational. As a result, networking infrastructure was developed in a much more bottom-up fashion: given individual machines administered by separate parties, the challenge was to knit together a facility that could simply route opaque packets between a suite of independently-developed black-box programs (and indeed, independently-developed networking software!), aiming for coarsely efficient performance globally, but optimizing for the ability to tolerate unpredictable workloads and failures across the system. Because these systems were only as useful as the number of nodes they knit together, they eschewed complexity in favor of simple interfaces that changed very slowly.

As data-centric services are increasingly tied into networking infrastructures, these two design regimes are moving rapidly closer together. A variety of emerging applications and computing environments call for dataflow schemes that couple the best of both these worlds: intelligent semantics-and-statistics-driven design from relational dataflow, coupled with the federated, semi-opaque, fault-presuming designs of networking systems. We will discuss a number of such applications and environments below.

1.2 Telegraph: An Adaptive Dataflow System

The above observations have led us to take a new approach to the development of infrastructure to support the emerging generation of data-intensive networked applications. Simply stated, our approach is to build the system around the movement of data in a complex networked environment. That is, rather than focus primarily on processing issues for a particular class of operators, we start by treating dataflow as a first-class component of our system, intended to serve as a substrate for an extensible and quite heterogeneous class of operators and applications.

Telegraph is an adaptive dataflow system that provides an extensible infrastructure for handling both finite and never-ending streams of data, coupled with encapsulated adaptivity features that can learn and react to un-
predictable environments without requiring complex standards or APIs. The goal of Telegraph is not to outdo traditional database or networking systems at traditional tasks. Rather, the goal is to address (and invent!) applications and computing environments that are not well-served by either of these traditional dataflow paradigms.

This paper presents the motivation and design of Telegraph, along with some experiences from our initial version of the system. We begin with a number of example scenarios, highlighting the limitations of traditional systems in serving these scenarios (Section 2). We then describe the Telegraph architecture, introducing basic concepts and components (Section 4). We continue with a description of a variety of different prototype applications that we have built to date in the initial version of Telegraph (Section 5), focusing both on the benefits that Telegraph brings to the development and performance of these applications, and lessons we learned in developing these initial prototypes. We provide an overview of related ideas and research in the area (Section 6), and conclude with some of the research opportunities we see in this area.

2 Three Emerging Application Domains

In this section we describe three emerging application domains that call for adaptive dataflow; we also describe our initial uses of Telegraph in each domain.

2.1 Sensor Networks and Ubiquitous Computing

Perhaps the most significant shift of computing platform today is the movement to so-called “ubiquitous computing”, in which tiny, pervasive devices will be deployed within everyday objects to carry out various tasks. While there has been a lot of work on data management software for handheld and other mobile devices, there has been little work on the dataflow infrastructure for ubiquitous computing.

An important data management problem in this arena is to support enormous numbers of sensors. It is now reasonable to assume that sensors will be deployed in increasingly pervasive ways, as dust motes [3], mixed in
with paint, and (in a less science-fiction mode) deployed in commonplace objects like food packaging, shipping boxes, and so on. These devices will have the ability to provide data via radio and optical networks in small regions with limited bandwidth; in aggregate, however, the data volumes are vast.

The networking community has begun to study sensors in earnest [4]. They have already raised some interesting problems that are reminiscent of database queries – in particular, the problem of returning aggregate results from a set of sensors with minimum communication, given that the sensor network routing topology is subject to frequent change. It is clear in this environment that more query and analysis functionality could be introduced into these networks.

In Telegraph, we have begun entering this area by supporting an application that combines data streaming in from traffic sensors on a freeway near Berkeley [5], joined with Internet cameras and accident reports fetched from the web. A screenshot of the application is shown in Figure 2.1, showing speed readings from the sensors, webcam feeds (the camera icons control the video choice), and traffic reports (the other icons on the highway). This combination of streaming data and fetch-based querying led to some of our basic design decisions discussed in ??, and also illustrated the application value of making used of multiple kinds of live data sources simultaneously.

This area is wide open for research, and an enormous number of questions exist; we list just a few here. At the most basic level is the question of how to interface sensor dataflows that have very different data movement properties – we will discuss this at length in Section ?? There are questions about how – and whether – to inject database-style code into the sensor network to perform data analysis close to the data production. There are also myriad questions about how to wed issues of dataflow, routing, and query optimization together in order to efficiently handle power management and network performance in the face of volatile network properties, noisy and missing data, multiple users, and continuous [7] or online [7] queries.

2.2 Federated Facts and Figures on the Internet

The deep web is a term that is increasingly being used to describe the variety of databases and services that are available in web browsers via methods other than hyperlinks. The deep web is estimated to be over 400 times larger in data volume than the traditional hyperlinked web [7]. Because of the lack of inbound hyperlinks, this data is not crawled by typical search engines, though recent work has begun to address this challenge [8]. Another problem with this data is that a sizeable fraction of it is not composed of full-text documents, but rather of structured data – “facts and figures”. Information Retrieval techniques are not typically apropos for facts and figures, which are better analyzed and browsed with database-style query facilities. Finally, the deep web is typically represented as “dynamic HTML”, and hence has not benefited from the service guarantees offered by web-caching business like Inktomi and Akamai.

The challenge of accessing this data with database-style queries has received a good deal of attention in recent years. Most of the attention has been on the challenges of writing relational wrappers [7], and on extending traditional DBMS functionality to work in an evolutionary fashion over those wrappers [7]. In Telegraph, as in some other projects [7, 9], we leverage the work on wrappers, but focus more directly on novel approaches to the problem of running queries over the unpredictable resources of the deep web.

In order to raise public consciousness about the possibilities of database queries on the deep web, we built a publicly-available web-based application in Telegraph studying campaign finances for the 2000 Presidential Election. This received a bit of local media attention in our area, and was visited by thousands of users in the month before the election. The application joined data from the Federal Election Commission and a variety of commercial sites offering demographic and personal information. Pre-composed queries were provided to find out statistical information about donors to the campaigns, including their home values and demographic information about their neighborhoods; we also allowed users to join the lists of donors with lists of famous actors (Gore received many more donations from well-known actors than Bush), and more recently lists of presidential pardonees (Gore received more donations from them as well). The site remains available as a demonstration at http://telegraph.cs.berkeley.edu.

A number of systems challenges were highlighted in our work on web queries, and are being addressed in the Telegraph architecture. As described in [7], dataflow optimization needs to adapt to unknown and volatile selectivity and cost estimation. Some web-based queries run for a very long time, and hence online query processing [7, 10] is important. Moreover, graceful fault-tolerance is important for these queries as well – because they run for a long time, they often fail, and starting them over is quite unpleasant. We address all of these issues in our discussion below.
2.3 Peer to Peer

Peer-to-peer systems like Napster and Gnutella represent a grass-roots emergence of widely-deployed federated databases. And yet these are very low-function systems from a database perspective – indeed, they are often referred to as peer-to-peer networks, and the only query they support is typically a substring match on filenames. Despite (or because of) the relatively low-tech designs of popular systems like Napster and Gnutella, the networking research community has eagerly embraced this area. Initial work has focused on distributed data structures to support equality lookups with few hops [?, ?, ?] – from a DBMS perspective this is an efficient distributed hash access method. Initial thoughts on database research in this space were also presented in a recent workshop paper [?], focusing on storage issues.

We see a huge research opportunity here in understanding how to inject more functionality into these systems. By augmenting these peer-to-peer “access methods” with extensible, adaptive dataflow, many new and useful applications can be introduced. The challenge is to do so in a manner that matches the grassroots nature of the systems. This means (a) working within existing APIs, or at least identifying the minimal algebraic needs to enhance those APIs, and (b) deploying attractive, “viral” applications to convince users to adopt enhanced versions of peer software with new functionality. This is quite a different exercise than the standards-driven language design typically addressed in the database community.

As an example of an interesting new feature, we have configured a version of Telegraph that serves as a Napster proxy. It intercepts requests from standard Napster clients, and forwards standard Napster searches and download requests on to multiple independent Napster-compliant servers – in this way, it serves as a metasearcher over these servers. In addition, it intercepts search strings with particular keywords, and translates them via dataflow queries into more interesting searches. For example, a search for a file entitled “artist=Beatles album=The White Album” is intercepted by Telegraph, which initiates a join between the CDDB album database and the union of the servers. CDDB is used to find the list of song titles on the Beatles’ “White Album”, and each of the Napster servers is queried for those songs – essentially a traditional index-nested loops join over multiple indexes. In the end, the Napster user receives a list of all the songs from the Beatles’ White Album available at any of the servers. We are toying with the idea of extending this via our own client application that queries our Telegraph-based proxy and automatically downloads a good choice for each matching song. For example, you could issue the query “Beatles”, and return some hours later to find the entire Beatles catalog stored on your computer, organized in directories by album. This involves not only a rather unusual query (with tricky duplicate elimination rules to pick a file for each song), but also management of the download process and exception handling for possible failures (choosing alternate copies of the song).

There are numerous challenges in this space, and we list a few that touch on dataflow issues. Our proxy design does centralized query processing, but an alternative design – e.g. one that builds on the ideas of [?, ?, ?] – might push more of the work of query processing into the network. As an example of this, consider a query on the usage of the system, e.g. to find the 10 most popular files over the last day – in Napster this information is available at the servers, but in more decentralized designs like Gnutella it is not. This bears some resemblance to the issues raised in the sensor network environment. It is an open question whether pushing processing into the peer-to-peer network is a good idea, but if it is to work then clearly inventive query processing techniques will be required to harness the scale and unpredictability of a peer-to-peer network. Before that can be done, peers have to be upgraded to be able to perform such work – enabling this requires defining a very minimal API that could be supported by thin peer software, and “selling” that API via attractive new functionality. Our Napster proxy is intended as a ”Trojan horse” to get our technology into this space; we hope to deploy it in the dorms at Berkeley next year and begin experimenting with more aggressive ideas and possibly the distribution of enhanced desktop peer software based on Telegraph.

3 Design Principles for Adaptive Dataflow Systems

maybe we start with more here on design goals of nets and QP?

- plan for a volatile world. faults are just ”really slow” machines, and you need unified mechanisms for load balancing and fault tolerance.
- design to work for both finite and infinite streams
fine-grained pipelining important for HCI, and also for system introspection. This is perhaps the only critical design constraint for dataflow operators.

loose coupling and thin APIs – you often need to learn, rather than communicate. Assume API extensions will evolve only for a killer app.

Minimize the lengths of “decision atomicity”. Decision per query hopeless for CQ. Same with decision per complete flow (i.e. per operator). Even decision per tuple dangerous, unless you can either guarantee the processing time of a tuple (problematic!), or guarantee preemptive scheduling of other work.

Allow semantics to be exposed extensibly, for exploitation in optimization. keep in mind that semantics over infinite streams (a la SEQ, TSQL) likely to be different than over finite streams (a la relational algebra). Hence need flexible APIs in the system to be able to take advantage of semantics.

In short, want adaptivity at multiple levels – the opaque point-to-point dataflow level, the multi-host load-balancing level (taking into account knowledge or observations of multiple resources), and the semantic level, taking into acct operator semantics.

4 The Design of Telegraph

Telegraph is a dataflow engine targeted at data-intensive applications that run in unpredictable environments. It consists of a core API for defining and executing dataflow operators, a set of pre-packaged operators that provide adaptive optimizations and fault-tolerance, a set of pre-packaged operators for online relational query processing, support for continuous queries, support for distributed process migration, a set of pre-packaged data access operators, and catalog facilities to support the storage and naming of data sources.

Telegraph can be configured in a number of ways, as illustrated in Figure ??

Database query engines typically present their dataflow API in a manner simply described as a superclass object from which all dataflow operators inherit basic methods; henceforth we use the term operator to refer to a generic instance of this superclass or its subclasses. Operators pass data – or handles to data – via these methods in a generic manner, without regard for the particulars of the specific operator subclass. In principle, this encapsulation allows most database query engines to be extended with new operators via such an API, though in many systems this API is not available except to engine developers.

A traditional dataflow model in database systems has all operators inheriting from an superclass called iterator, with a constructor to initialize state, a method next() that returns a new data object, and a destructor that cleans up state. Operators pass data – or handles to data – via these methods in a generic manner, without regard for the particulars of the specific operator subclass. In principle, this encapsulation allows most database query engines to be extended with new operators via such an API, though in many systems this API is not available except to engine developers.

A traditional dataflow model in database systems has all operators inheriting from an superclass called iterator, with a constructor to initialize state, a method next() that returns a new data object, and a destructor that cleans up state. Iterators are “connected” into a flow graph by object pointers that allow consumers to invoke the methods of their producers. Implicit in this iterator model is a threading model: an entire flow consists of a

1The distinction between passing data objects or handles depends on whether one is in a single address space (in which case handles are typically cheaper since they avoid copies), or in two address spaces (e.g. on separate machines, in which case the data is typically shipped directly.) For the discussion below, we will not explicitly speak of passing handles rather than data, but within a single address space we will assume that handles are being used as appropriate.
single thread, in which operators “pull” data along the flow by synchronously invoking the next() method of their input producers, with (handles to) data objects returned on the stack. We refer to this as the “synchronous/pull” model, since control flow is via synchronous procedure calls, and dataflow is done by consumers “pulling” records from producers on demand.

Unfortunately, the synchronous/pull model is poorly matched to external data feeds from sources like equipment monitoring sensors, which may asynchronously pump out data streams. An alternative approach—typically used in networked systems—implements an operator superclass that (for lack of a better name) we call data-pump, which generates a separate thread for each operator instance. Datapump operators are connected into a flow graph by producer/consumer queue objects; operators consume data from their input queues, and place result data on their output queues (which are in turn the input queues of operators “downstream”). We refer to this as an “asynchronous/push” model, since control among the various operators is asynchronous, and dataflow is done by producers “pushing” data onto queues without waiting for requests.

In Telegraph, we attempt to bridge these worlds, providing an asynchronous/push API that can be configured to mimic a synchronous/push API. We need both models in Telegraph: asynchronous/push to handle streaming examples like the sensors mentioned above, and synchronous/pull to handle sources that only respond to explicit “lookup requests” (e.g., web forms). Another use for synchronous/pull is with battery-powered wireless sensors like Smart Dust motes [?], which may run in low-power “sleep” mode except when periodic requests to pull data arrive. A dataflow in that case should pull (“sample”) only occasionally from such a source, depending on a user’s need for resolution (e.g., as determined by a user’s gestures at a graphical interface). To see how Telegraph can provide synchronous/pull behavior in an asynchronous/push API, it is useful to carefully examine these models, which are traditionally thought of as being quite distinct.

Consider a typical synchronous/push API, in which each datapump operator has its own thread, and small bounded queue objects exist on every dataflow edge in the flow graph. We study a small example flow of two operators, with a very fast operator (the “producer”) flowing data to a very slow operator (the “consumer”). The disparity in speeds guarantees that the queue fills shortly after the flow begins. Indeed, in steady state, the flow behavior can be described as a repetition of the following pattern:

1. The consumer dequeues a single record and begins to process it.
2. The producer immediately enqueues a record, and is unable to enqueue any more records on the full queue.

Note that this behavior is very similar to that of a synchronous/pull system: control is passed from consumer to producer in a nearly synchronous fashion, with the consumer operator “driving” the scheduling and dataflow of the producer operator, one record at a time. The only dataflow distinction is the very brief parallelism that occurs after the consumer dequeues and begins processing, while the producer quickly enqueues a new record.

Having observed that an asynchronous/push model sometimes looks very much like a synchronous/pull model in this extreme case (fast producer, slow consumer), we now show how to achieve this effect in general. In Telegraph, the queues between operators are embodied in instances of a class called Connector; connector objects do not have their own thread, but rather have methods “connector.put” and “connector.get” that are invoked by the two operators on either side of the connector. By default, Connectors are queues, with connector.put serving to enqueue, and connector.get serving to dequeue; this provides a straightforward asynchronous/push system. However, PullConnector is a subclass of Connector that forces pull-like behavior by “overloading” the put and get methods. A producer’s Connector.put call on a PullConnector blocks by default, until a consumer’s matching Connector.get call is made. In essence, a PullConnector is like a queue of length zero; it may also be thought of as a “handoff semaphore” for two threads, allowing them to proceed only when both threads are blocked at the connector. This provides the pull-like behavior seen in the previous example of data pump operators with mismatched speeds—even for operators with a variety of speed relationships.

A common criticism of the asynchronous/push model is that it requires significant system overhead in threads and thread scheduling, whereas the iterator model requires only a single thread for an entire query plan. This criticism was typically made in the context of older operating systems in which a thread of control was associated with a full process context; in modern operating systems with lightweight threads one would expect that this overhead would be negligible. However, we found in our first implementation of Telegraph in Java on Linux that threads are indeed expensive on that platform [?, ?]. As a result, we include our own scheduler in Telegraph, and program our datapump operators to yield control to the scheduler at natural points, in an event-based programming style like that of [?, ?]. Traditionally, programming with yields is considered more difficult than programming
straightline code in a preemptive threading system, but we have found in typical dataflow operators that the choice of the points for including yields is actually quite straightforward. In any case we consider this to be a detail rather than a fundamental issue; we expect that a system with a mature thread package would render the use of data pumps with their own threads perfectly viable, and we hope to migrate to such an environment soon.

It is also worth noting that a single-threaded plan is often inappropriate in the Telegraph context for another basic reason: adaptivity. Adaptive operators like eddies or Xjoins [?] require the ability to take action while an upstream operator is busy (or stalled, e.g., on a network connection) trying to produce its next record. In our work on eddies, we achieve this by providing a thread per operator; this was done in a C-based implementation on Solaris, and was found to have acceptable performance. Our work on Xjoins and query scrambling [] was done within a traditional synchronous/pull iterator-based query processor (that of the Predator system [?]). In that case, multi-threading had to be introduced into plans in unusual ways; for example, the Xjoin itself was a multi-threaded operator, and this was a non-trivial extension of the simple iterator infrastructure originally provided.

We close our discussion of the Telegraph dataflow API by addressing one more commonly-cited attractive feature of the synchronous/pull model: its by-now familiar iterator programming model, which is commonly taught in database textbooks. This programming model can be provided with a bit of code rewriting in Telegraph. To achieve the mapping, an iterator-style implementation would use the operator.next call, rather than the operator.get call; a default operator.get call simply calls operator.next, and uses the return value to invoke connector.put on the output queue and yield the processor. Programmers of iterator-style operators are unlikely to insert their own yields into their code, which means they will (like traditional iterators) “hog” the processor in the current version of Telegraph until they produce a tuple. While this seems like a natural and acceptable penalty for programming in an iterator style, note that even this penalty would disappear in a preemptive thread-per-operator implementation of scheduling.

4.1.1 Specifying Flows in Telegraph

In addition to defining an API for operators, one also needs ways to specify flows and execute them. In Telegraph, there are currently two different ways to specify a flow for execution. For relational queries, Telegraph supports a simple subset of SQL suitable for select-project-join queries with aggregation, grouping, drill down, and online query processing facilities. Telegraph’s parser translates these queries into a dataflow structured around an eddy, as described in Section XX, with standard (push) connectors at the edges.

Telegraph also supports an interface that allows users to explicitly describe a flow graph of Java operator objects and connectors via a simple scripting language, and to execute that flow graph by passing a script to Telegraph. We use this second, more explicit interface, to allow us to experiment with new application domains, without the need to define a syntax and parser for a new declarative language. This interface is in keeping with the oft-cited goal [] of developing toolkits for database research, rather than closed, end-to-end systems. We are experimenting with this interface for a number of issues, including the use of a mixture of push and pull connectors for sensors. We call dataflows with a mix of push and pull connectors “Fjords”; these are discussed in a companion paper.

4.2 Juggling Eddies: Operator and Data Ordering

One of Telegraph’s key design goals is to provide efficient, robust flows for long-running and continuous queries, even in unpredictable environments like wide-area and sensor networks. Meeting this challenge requires flexibility in constructing and maintaining dataflow graphs. In a traditional database system, there is a query optimization phase before the flow starts, which compiles a declarative query into particular set of operators, connected in a particular graph topology. This static compilation approach does not meet the design goals of Telegraph, since it does not provide any opportunity to change an ongoing flow graph in the face of changes in the environment. This can be problematic even in the case of relatively short queries in a volatile performance environment []; it is a serious problem for continuous queries, which by definition run far longer than the rate of change of any environment.

Instead of static optimization, Telegraph uses eddies to continuously adapt the dataflow routing among operators [?]. An eddy is itself an operator – i.e., the eddy object class is a subclass of the generic operator superclass. One eddy is intended to be interposed between all pairs of operators in a flow, as shown in Figure ??; Telegraph’s

---

2 Telegraph can of course execute flows without eddies, which are processed as standard dataflows.
SQL parser constructs dataflows in this manner by default. As originally described in [1], the eddy uses its central position to serve as a tuple router, allowing it to effect different orderings of the operators in the flow, by routing tuples through the operators in different orders. In essence, eddies cast the traditional query optimization problem as a tuple routing problem in a dataflow.

In Telegraph, we have enhanced eddies in two important ways beyond the description in [1]. First, we extend eddies to correctly run a query that contains multiple redundant operators – for example, two different access methods for the same data source, or two different algorithms for the same logical operation. In this scenario, the eddy serves not only to order the operators, but also to direct competition among redundant operators. Telegraph’s SQL parser constructs a flow with an eddy at the center, multiple alternative access methods for each data source, and multiple alternative join algorithms for each logical join. By handling both operator reordering and operator competition, eddies translate the full query optimization problem into an online routing problem that can be addressed in a continuously adaptive fashion. There are some subtle details needed to ensure the correctness of the query, which are described in a companion paper – these center around a mechanism to allow competing operators to share state [2]. We have only begun our study of efficient eddy routing policies; this problem appears to be quite related to the $n$-armed bandit problems studied in the control theory and machine learning literature [2]. Although these policy questions remain open, we shall see examples in Section XX that illustrate the substantial benefits of the competitive eddy mechanism; these examples lead us to believe that the mechanism is useful, and that routing policies are an important topic for further study.

Our second enhancement is to use online reordering of records [3] to manage the eddy’s input queue, so that the eddy can buffer multiple records and choose which of them to route next based on a controllable policy. As noted in [3], users in long-running or continuous query environments can derive significantly more satisfying performance if they can prioritize the delivery of tuples. An open problem in that paper is to decide where to place reordering operators in a traditional, static query plan; by integrating reordering into an eddy, we solve that problem by allowing tuple reordering between each operator invocation. The ability to do tuple reordering gives the eddy full control over the dataflow ordering, both in terms of the priorities of tuples in the flow, and the order and selection of operators visited by each tuple. The combination of these two distinct reordering capabilities gives the eddy a great deal of flexibility in producing a result stream that matches the changing and perhaps exploratory needs of a user. In a companion paper we demonstrate the particular benefits of this combination for generating interesting partial result records – projections of full result records – in a particularly efficient pipelined manner. Surprisingly, it can be shown that reordering tuples in an eddy can provide benefits beyond those available by the best placement of reordering operators in a static plan.

4.2.1 Extensible Optimization With Eddies

Eddies also serve as a point of extensible optimization in Telegraph. As described in the original paper on eddies, each record that flows through the eddy is associated with a small bit mask, with two bits for each operator connected to the eddy. The first bit is called the ready bit, and should be set to 1 when the record is eligible to be sent to the associated operator. The second bit is called the done bit, and should be set to 1 when the record should never again be sent to the associated operator. In Telegraph, these bits are maintained by an object called an EddyHelper, which is associated with the eddy. The EddyHelper is parameterized by rules that constrain the setting of the ready bits – corresponding to constraints on legal operator orderings. For example, a record would not be sent to a grouping operator – i.e. that operator’s ready bit would remain at 0 – until the done bits were set to 1 for all join and selection operators. Similarly, a record would not be sent to an $R \bowtie S$ operator unless its done bit were set for either an $R$ access method (meaning it contained attributes from $R$) or an $S$ access method (meaning it contained attributes from $S$). This is akin to the “avoid Cartesian products” rule commonly used in traditional query processors, which avoids combining a record from one relation with records from other relations unless a join predicate is available to form a small combination. Chop that?

In a similar fashion, the EddyHelper is parameterized by rules that control the setting of the done bits in intermediate records – corresponding to avoidance of redundant work. For example, in an eddy with 3 competitive operators for joining $R$ and $S$, once a tuple is returned from one of the operators, the done bit would be set for all of the operators. It is also possible to have semantic rewritings that allow multiple operators (e.g. cartesian product and selection) to compete with another choice (e.g. a join), via appropriate EddyHelper rules. In a companion paper, we provide more details about the relationship between routing policies and legal relational query plans [2].

When a new operator is introduced into Telegraph, it should not be connected to an eddy in a flow until the
appropriate ready bit constraints have been added to the catalog, so that the EddyHelper can ensure that only correct routing optimizations occur. Until that time the operator can simply be placed explicitly in the correct place within the flow, as shown in Figure xxx, possibly by the use of multiple eddies in the plan for clusters of operators that are subject to optimization. This approach contrasts with that of earlier extensible optimizers like Exodus, Volcano and Starburst, all of which used rules to allow particular transformations, rather than to constrain them. In the eddy context, it seems to us more natural to assume a default of extreme flexibility in routing with annotations for constraints, rather than a default of extreme inflexibility annotated with legal rewritings. However, our experience with this extensibility interface has been frankly limited to date, and we intend to both formalize and revise this interface in future work.

4.3 FLuX

Telegraph is intended to run over both finite and never-ending data streams. During the course of an infinite data stream, as in a continuous query or sensor scenario, even the most reliable components will fail. Even for long-running but finite queries, it can be frustrating to have to restart on failure. Hence we considered fault-tolerance to be an important goal for Telegraph. As we shall see, we can take advantage of the dataflow programming model in Telegraph to introduce some novel optimizations to traditional fault tolerance schemes.

In Telegraph, we encapsulate fault tolerance and subsequent load balancing within a single encapsulated operator that we call FLuX: Fault-tolerant Load balancing eXchange. As its name suggests, FLuX is an enhancement to the Exchange operator of Volcano. FLuX provides the same facilities as Exchange for taking a data stream and repartitioning it among many participating machines, and is deployed in the same fashion – a replica of the full plan runs at each node, with inter-node communication encapsulated within the FLuX replicas, which partition their incoming data among both their local output queue and remote FLuXen. FLuX removes from Exchange the push/pull interfacing that we handle instead with our interface in Section XXX. In order to introduce fault tolerance into the scheme, FLuX introduces the following features missing from Exchange:

- **Adaptive flow partitioning**: Unlike Exchange, FLuX does not use a static partitioning to send data to different machines. Instead, FLuX can dynamically adapt its partitioning so that it sends each recipient a fraction of data that matches the recipient’s relative speed. This ensures that machines with heterogeneous hardware or available resources are given no more than a fair share of work to do at any time. This idea was presented earlier in the River system, where it was shown to provide effective load balancing for parallel sorting in the face of dynamic machine behavior.

- **Repartitioning of accumulated state**: A limitation of River was that it only adapted the flow partitioning, but left previously accumulated records in the locations where they had been received. This is problematic for partition-based operators like hash join, which require all tuples within a particular partition to be collocated at all times. FLuX provides an interface that allows operators to ensure that previously-received records are relocated (or “shed”) to the machine that will receive future records of the same kind.

- **Replication of streams**: Unlike exchange or River, FLuX can send a replica of each record to one or more additional machines, to provide fault tolerance. In essence, FLuX provides this feature for transient state in much the same way that chained declustering provides it for stored relations.

Two standard mechanisms for fault tolerance in systems are checkpoints and process replication (“process pairs”). FLuXen can be used as a distributed checkpointing mechanism in a straightforward manner: the recipient of a replica record simply stores it to disk for subsequent use in recovery. FLuXen can also be used as a form of loose process replication. In this scenario, replica records are passed along the flow at the receiving machine. However, these loosely-replicated dataflow operators are not as strict as process replicas – we need not enforce that the operators on two separate machines maintain exactly the same program counters, heap memory, and so forth. This is because the only properties visible across machines in a dataflow system are the streams of data communicated over the network. Therefore each machine can run at its own rate and manage its own memory as it sees fit, as long as it maintains some stream-correctness invariant at its output; we will briefly discuss these invariants shortly. There is also a middle ground between checkpointing and process replication using FLuXen. It is possible for a machine to choose to store some of replicas to disk, while passing others directly along the

---

3We use a plural akin to that used for “VAX” or “ox”.

10
dataflow. This amounts to a middle ground between lazy replication of computation (a la checkpointing) and eager replication of computation (a la process pairs). Lazy replication leads to longer recovery time, whereas eager replication consumes more resources in the no-failure case, but can result in instant recovery. We intend to study this spectrum as it relates to dataflow in subsequent work. In the meantime, the FLuX architecture provides us flexibility to experiment with these parameters, while also guaranteeing correctness of flow results. The details of the correctness proof, including a discussion of sequence numbers and duplicate elimination, appear in YYY; they are beyond the scope of this paper.

With regard to stream correctness invariants, these depend both on the application in general, and on the next operator in the (parallel) pipeline in particular. Without additional knowledge of the application or operator semantics, a stream can be considered to be correct only if it is identical to an “ideal” stream – the stream that would have been produced without any faults in a system without replication. However, some applications or subsequent operators might relax this requirement. For example, an application with best-effort semantics might have a stream correctness invariant stating that output streams need to be subsequences of the ideal stream. As another example, a set-oriented semantics might define stream correctness in an order-independent, duplicate-insensitive fashion – this would occur, for example, if the next operator in the stream were a duplicate elimination operator in a strict relational query. The definition of such invariants, and their relationship to various query semantics and query operators, seems to be interesting area of future work that ties systems issues directly to semantic issues. Clearly an extensibility interface is required for this linkage to be made flexible, but we are postponing such an interface until we have experimented with a variety of different application scenarios.

In the face of machine failure, extra work must done by other machines in order to mask that failure. As in fault tolerant storage schemes like chained declustering, we would like an adaptive load balancing technique, so that the burden of tolerating the fault is spread thinly across the machines, not focused at the “buddies” of the failed machine. In fact, as described in [?], adaptive load balancing is an important feature even in a fault-free parallel dataflow system. In Telegraph, we enhance the adaptive flow balancing work from River to correctly manage partition-based operators. FLuX contains an API that allows an operator to shed partitions from one machine to another. FLuX also enhances River with a load monitoring facility to detect when machines change their workload balance. When such a change is detected to be large and persistent, coordinating FLuXen on various machines may choose to notify their receiving operators of the opportunity to shed state. This division of labor does require that partition-based operators receiving from FLuXen notice this opportunity, and take advantage of the FLuX API to perform shedding. We are currently refining designs for a minimally intrusive shedding API between partition-based operators and FLuXen; we also examining adaptive policies for deciding when and how much load to shed. In Section XX we demonstrate some promising initial results.

### 4.4 Online Query Processing

Telegraph currently includes a number of online relational query processing operators – both for online aggregation, and for producing incremental results to standard enumerative (i.e., aggregation-free) queries. We include a variant of the Xjoin [?], which produces pipelining results, and which can perform efficiently in the face of delayed sources. We also include an asynchronous index join, which is able to issue multiple concurrent, asynchronous requests to an indexed source, while continuing to fetch from the outer relation; this is important for high throughput in the web query context, as described in [?]. Via an enhanced eddy routing policy, we are also able to broadcast a tuple to multiple indexes simultaneously and combine the results correctly, another feature cited in [?]. We provide simple loop-based ripple joins for non-equi-join predicates. We also include selections, projections and grouping, with grouping done via hashing in an online fashion as suggested in [?]. Because many of our initial demonstration applications were targeted at web queries, we include outer join variants of all our joins as well, since outer joins are particularly useful for exploratory queries over heterogeneous schemas. We currently use very simple estimators for online aggregation queries and provide no built-in confidence intervals, in part because we have no way to ensure that the data streaming from web sources arrive in random order. This is a topic we intend to explore in future work.

In addition to our work on online query processing, we are researching the possibility of returning partial result records – that is, projections of actual result records, which may be available for display earlier than the complete result records. This research is described in a companion paper.
4.5 Supporting Continuous Queries (CQ)

For applications that incorporate streams of data, it is often natural to process the data using some form of continuous query (CQ) \[?\]. Unlike traditional queries, which run over a particular database state and terminate once they have completely processed that database state, continuous queries produce results as the database changes, and run until they are explicitly terminated. For example, in a sensor-based application such as highway traffic monitoring, a continuous query could be used to measure the average flow of cars in a particular lane over time. In an information dissemination application, continuous queries representing user interests can be used to filter a stream of incoming documents or tuples so that they can be routed to the people who need them.

Continuous query processing has been the subject of increasing attention in the database community recently \[?, ?, ?, ?\]. The major challenge addressed in this work is scalability, particularly in terms of the number of queries that the system can process. For many CQ environments, it is expected that hundreds of thousands or more queries will be active simultaneously. These queries are processed incrementally as the data flows through the query processor. It is imperative for the query processor to be able to keep up (over time) with the arrival rate of the incoming data stream. In order to achieve this goal in the presence of large numbers of queries, CQ systems typically employ some form of multiple query optimization and attempt to share common query subexpressions and structure.

Since Telegraph has been designed from the ground up to support adaptive dataflow, it provides a number of interesting options for implementing highly-scalable CQ processing. We are currently investigating two approaches: 1) A fairly traditional CQ approach in which the queries are indexed, and 2) a more novel approach that exploits the inherent adaptivity of the Eddy operator. We briefly discuss these approaches below.

4.5.1 Index-based approach

In some ways, a CQ system can be thought of as a traditional database system stood on its head. In a traditional system, the database holds a large number of data items (e.g., tuples). When a query is submitted to the system, the query processor attempts to locate the relevant data items as quickly as possible. Typically, this process is aided by indices. The indices allow the query processor to disregard data items that not relevant to the query. In a CQ system, the situation is reversed. A large number of queries are stored in the system, and when a new data item arrives, the system must quickly match that data item to the queries for which it is relevant. Not surprisingly, the mechanism employed for solving this matching problem is similar to the traditional one, namely the queries are indexed for fast matching to incoming data items \[?\].

A recent example of the Index-based approach is our XFilter system \[?\] for matching XML documents to standing XPath queries. In XFilter, the XPath queries are converted to Finite State Automata and indexed using a hash-based index. In on-going work, we have converted the system to use a Trie-based index in order to allow sharing of common query prefixes. Systems that support other types of continuous queries (e.g., SPJ-based) such as NiagraCQ \[?\], as well as active database systems also employ query indexes.

In Telegraph, this matching functionality is encapsulated in an operator, which can then be attached to a data flow. In effect, the matching operation is treated like an Indexed Nested Loop join (albeit, a sophisticated one), where incoming data items are joined with the queries by probing the query index. The default asynchronous/push-based flow mechanism of Telegraph is well-suited for CQ systems. Furthermore, the ability to build hybrid flows that include synchronous/pull-based processing allows the system to handle Continuous Queries that access a mix of streaming and database-resident data.

4.5.2 CACQ - Continuously Adapative Continuous Queries

We are also exploring a more novel approach to CQ processing, which exploits the continuous optimization capabilities of the Eddy operator. In this approach the original Eddy implementation \[?\], which was intended to run a single query, extended to handle multiple queries. If a group of queries share data streams, then the modules of all those queries can be connected to the same Eddy. To handle this case, the bitmaps indicating which modules each tuple must pass through are extended to include the modules of all of the running queries.

Extending the Eddy operator in this manner allows queries that access a particular data stream to share a single scan of the stream, saving on copying overhead. Furthermore, query operators that are similar can be combined into a single module. For example, comparision predicates that differ only in their constants can be combined, much as such predicates are combined in systems like NiagraCQ.
This arrangement has several potential advantages. First, it provides for efficient processing of multiple queries by allowing input data streams to be efficiently shared among queries and by enabling similar operators to be combined. Second, the reordering of operators inherent in Eddies allows for increased sharing of processing effort among multiple queries. Systems based on more traditional notions of query plans (e.g., NiagraCQ) cannot easily reorder the execution of their operators. Therefore, these systems can combine similar filters or joins only when they appear at identical locations within query plans. In contrast, with Eddies, tuples contain state information that indicates the operators that they have already visited and those that they have yet to visit. This mechanism allows tuples to take varying routes through operators, which enables the system to opportunistically take advantage of shared processing.

It is important to note that these two approaches to CQ processing, Index-based, and Continuously Adaptive are not mutually exclusive. Thus, we are implementing both of them in Telegraph. Initial results are promising, but a detailed performance study will be needed to determine how best to use these mechanisms for the various types of continuous queries that will be used by different applications.

4.6 Operator Migration: Towards Distributed Telegraph

4.7 Data Access Operators

4.8 Catalog Management, XML, and Introspection

Like most database systems, Telegraph stores the meta-information describing accessible (wrapped) sources and their properties in the catalog. To simplify catalog management, we chose to store the Telegraph catalog in XML format which is parsed according to a description file. This file, similar to a DTD, describes the schema of the catalog and includes the column types for catalog tables.

There are a couple advantages to our approach. First, this approach is more amenable to catalog evolution than traditional hard-coded catalogs. By leveraging the Java introspection features, we can construct various versions of the catalog data-structure, Catalog, from simply the XML catalog file and its description file. For instance, imagine that we wanted to store a logo with each table in the catalog, which Telegraph might use to present a graphical interface to the database. This change can be incorporated by appending a column of type java.lang.Image to the TABLES table definition in the description file. Introspection allows the Catalog to instantiate an object of a user-specified type at runtime. Thus, when the Telegraph catalog parses the description file, it will automatically instantiate an image object for each logo listed in the catalog file. In cases like this, a modification to the catalog schema may not require recompilation or relinking the Telegraph server or dependent applications. Thus, we achieve better separation between versioning of the software from versioning of the data.

Another advantage of maintaining the catalog in XML, a standard, flexible, text-based format, is that it allows to leverage a variety of existing tools. We can edit the catalog using exiting editors without the need to design and implement a DDL. Furthermore, we were able to allow transactional access by layering existing software, Berkeley DB transactional store, on top of the catalog.

In conclusion, by encoding the catalog in XML, and leveraging Java introspection features and existing tools, we are able to modify and evolve the catalog more easily than traditional catalogs.

4.9 Building a Dataflow System in Java

Some good, some bad. Especially memory and thread management. See companion paper.

5 Some Initial Results

The applications described in Section ?? illustrate the broad utility of the Telegraph system in a variety of environments. But they are less effective at illustrating particular technical features of the system. In this section, we present a range of preliminary results of new features in Telegraph, running in single-site, parallel, and distributed modes. Space constraints prevent in-depth performance study, which we reserve for papers focused on particular aspects of the system. Our goal here is to provide a preview of some of the salient new features of Telegraph, as examples of the benefits possible via our adaptive dataflow techniques.
Figure 2: A competitive eddy of three join alternatives, compared against each alternative individually.

Figure 3: A close look at the first few seconds of the same graph.

5.1 Adaptive Dataflow on a Single Node: Competition and Eddies

The original work on eddies focused on their ability to reorder operators in dataflow. As described Section ??, we are studying the ability of eddies to also tackle another aspect traditionally associated with static query optimization: the choice of operators in a plan. Indeed, as we shall see, eddies can outdo static optimization decisions in certain cases.

To illustrate this, we study a query from our Election 2000 demo, which finds movie stars who donated to the presidential campaign of George W. Bush. It uses two web sources: the first is a list of movie stars from Yahoo, and the second is a database of Federal Election Commission campaign donations maintained by the FECinfo site at www.tray.com. We join the stars’ names to the donors’ names, and return the amount of donation, address, and occupation for each donor that has the same name as a movie star. The Yahoo movie stars appear on a single page at http://dir.yahoo.com/Entertainment/Actors_and_Actresses/Complete_Listing/, and hence are accessed from a TeSS operator as a stream of records, much like a disk scan. FECinfo offers multiple access paths for the donation data: one can receive all the data as a scan (we will call this FECscan, available at http://www.tray.com/cgi-win/xjtoc.exe?DoFn=P00003335BUSH,GEORGEW00), one can do index lookups by donor’s last name (FEClastIX), or one can do index lookups by (last name, first name) pairs (FEClastfirstIX – both indexes on the web at http://www.tray.com/cgi-win/indexhtml.exe?MBF=NAME). This leads naturally to three alternate query plans in Telegraph: a pipelining hash join of the two scans, an asynchronous index join of Yahoo with the FEClastIX index, and an asynchronous index join of Yahoo with the FECinfo.lastfirstIX index.

A fourth possibility is the competitive eddy that runs all three of these joins in competition. We made one optimization to avoid wasted work in the competition: whenever a donor is returned from an index lookup, we insert the donor into the hashtable of donors used by the hash join. Thus the competitive operators cooperate in accumulating state.

We performed a run of the query for each of the four plans described above. Figure 2 plots the number of records returned by each of these plans over time, this rate being a reasonable metric for an online enumerative query [?]. Figure 3 presents the same data, focusing only on the first few seconds of the query. Note that Figure 3 shows that the index join over FEClastfirstIX provides the best throughput at first, but that the hash join quickly catches up as its hashtables fill, producing greater expected numbers of matches per lookups. The index join on FEClastIX performs poorly, because the FECinfo site responds to such queries much more slowly; the reason for this is unclear, but highlights the need for adaptive selection of access methods. The competitive eddy scheme enjoys the advantages of all the joins for the first few seconds – the indexes find matches for every Yahoo tuple sent to them, and some of the Yahoo tuples sent to the hash join find matches there as well. As the hash join begins to dominate, the eddy detects the benefits of sending Yahoo tuples to the hash join, and increases the ratio of Yahoo tuples sent that way. In the long term, Figure 2 shows that the hash join is the best algorithm over all, and

\[^4\text{In fact, this is accomplished in an elegant fashion by exposing these hash tables as operators called STate Modules (STeMs). In addition to making the API more elegant, STeMs have a number of additional benefits, one of which is that (with some care) the hash table from the hash join can also be used by the index join as a local lookaside cache – STeMs deal with tricky issues of duplicate detection and elimination, which are beyond the scope of this paper. A thorough discussion of STeMs with preliminary performance results appears in [?].}\]
that the competitive eddy scheme does nearly as well as it, though not quite as well. The eddy cannot quite match the hash join over time because the eddy scheme continues to send a fraction of Yahoo tuples to the index sources, to check if they have changed behavior. This is a classic example of the "exploration vs. exploitation" tradeoff discussed in Machine Learning [?] : we have to choose between "exploring" the possibility that a once-ineffective operator has improved, and "exploiting" the observed benefit from the currently-effective operator. In this case, we pay a small cost for continuing to explore the index join options. Some tuning of the eddy routing policy might dampen this exploration a fair bit more, though this will make the system less quick to adapt to sudden changes in performance. Clearly this is an area for further study.

5.2 Adaptive Parallel Dataflow: FLuXen in Action

In our next experiment, we study how the FLuX operator balances load and tolerates a failure in a small cluster of machines. To illustrate this behavior, we ran a parallel pipelined hash join across four identical machines, and caused the system to fail on one machine. The machines were homogeneous PCs, each containing a single 666 MHz Pentium III CPU, 128 MB RAM, running RedHat Linux 7.1. The join was a primary-foreign key join on the sid attribute of two synthetically generated relations, student(sid, dept) and enroll(sid, course). Each student is enrolled in exactly five courses, and each row in both tables is around 100 bytes. The two tables were hash-partitioned across the four machines by their non-join attribute, to highlight the parallel communication in the FLuX. We ran the query long enough to observe the FLuX adapt to a single fault; we did not complete the join.

In the query plan, each node had one scan operator per relation which asynchronously fetched local tuples, and these scans were connected to a pipelined hash-join operator on each node via the FLuX operator. The hash-join operators contained 67 partitions that FLuX could move in order to balance load; each partition was replicated on one other machine. All the partitions fit in main memory during the course of our experiment.

Figure 4 shows the aggregate throughput seen across all machines during our experiment; the points represent throughput averaged at one second intervals. At $t = 60$ sec, we cause one of the four machines to fail on one of the machines, leading the FLuX operator to go into recovery mode – the goal of recovery mode is to bring the
system back up to a fault-tolerant state, meaning that it has to re-replicate all the data that was lost on the failed machine. As can be seen from the graph, this recovery mode is currently implemented as a quiescent operation. It took about 14 secs for the FLuX operator to make copies of partitions that were lost, and redistribute them amongst the remaining three machines. Once recovered, the throughput of the system drops quite a bit because the failed machine is no longer processing a share of the work – the average throughput before and after the fault is shown by the dashed horizontal lines. This reduction is somewhat more than might be expected – the average throughput goes down by about 42%, where is might be natural to expect a reduction of 25%.

To study the behavior more closely, Figure ?? shows the average queue lengths at the (merged) input of each of the join operators during this experiment. Throughout the experiment, the FLuX operator attempts to equalize these queues by moving join partitions and thus shifting its associated load. Note that the queue lengths are approximately equal, and if the lengths diverge, they subsequently converge again. Moreover, both at startup and after the failure, the FLuX operator does its best to equalize load. By examining the two graphs simultaneously, it can be seen that a load imbalance (as indicated by a low queue length in Figure 5) is quickly followed by a low-throughput period (Figure 4, during which the system sacrifices performance to migrate a partition. A troubling aspect of the graph is its fairly significant oscillation – this suggests that more tuning of our migration threshold is required.

5.3 Adaptive Distributed Dataflow: Moving Code to Data

In this section we present a study of quiescent and online migration in one simple “microbenchmark” scenario. We have two identical machines (Pentium III, 664 MHz, 256 KB cache, 128 MB RAM), running Linux 2.4.2-2, IBM JDK1.3 (jit enabled). The machines are connected by a 100Mb ethernet, but in order to see the effects of moving large operators across a long-haul network we simulate a slower network by padding our Telegraph message block to be considerable longer than necessary, resulting in much lower apparent bandwidth.

In order to focus closely just on the mechanisms of operator migration, we study an extremely simple query plan of one Log operator running on Machine A, which accepts a stream from a network and continually appends the contents of the stream to a disk. Machine A has additional load on it from other processes, but Machine B is unutilized – we model this by introducing a temporary sleep into the Log operator on Machine A that is absent on Machine B. We study the overhead of migrating the Log operator to Machine B either via Quiescent or Online Migration. We assume that the input stream is equally “close” to both Machine A and B – i.e., the throughput of the Log operator is dependent only on the load of the machine it runs on, not on the rate of arrival of the stream. This is also the reason we chose a Log operator for this study – it has no output destination to consider, allowing us to focus strictly on the migration mechanisms.
Figure 6 demonstrates the throughput over time of three experiments: no migration, quiescent migration, and online migration. At about 50 seconds into each of the latter two experiments, the migration begins. One can clearly see the effects of quiescing the connectors in the quiescent case: during the time of migration, the throughput of logging is non-existent. In the online case, there are a few issues to note. First, the online case temporarily exhibits higher bandwidth than the quiescent case because it continues to log while it migrates. A user observing the operator (unlikely in this artificial logging setting, but quite likely in an online or continuous query processing environment) might prefer online migration during that window. Second, the online migration scheme takes quite a bit longer to complete than the quiescent migration, ending around 110 seconds. One reason for this is that the online migration has to move more data than the quiescent migration: all the data that arrived before migration began, plus any data that arrived during migration. An additional reason is that online migration requires Machine A to continue logging during migration and hence perform migration under load (on its network connection, I/O bus, and CPU), whereas in quiescent migration it could focus exclusively on migration. However, once online migration completes and decides to kill the clone at Machine A, its throughput climbs at about the same rate as that of quiescent migration as would be expected.

This microbenchmark only shows the effects of migration mechanisms. We have yet to study policies for deciding when and how to migrate. We have also only begun to build our infrastructure for distributed monitoring, to allow the detection of beneficial, willing targets for migration.

6 Related Challenges and Related Work

6.1 Languages, Models and Semantics

Historically, DBMS folks have bitten off this issue alongside the systems issue. This may have been a good idea for performance optimizations, but it’s limited our applicability and attractiveness as extensible infrastructure code and attractive shareware (cite others).

That said, there are interesting challenges here, especially in the sensor space, but in other application spaces as well. See Praveen, TSQL & TimeSeries, sequence mining, etc. Moreover, by exploiting language semantics, a variety of system performance enhancements can be achieved – we have tried to expose APIs for some of these in anticipation, but by picking languages or at least concrete applications we expect to learn more lessons about the interweaving of semantic and system optimizations.

Note that online query processing is a temporal semantics for streams of tuples in a relational model! What is the role of approximate answers and partial results here?

One question is whether to go for query languages explicitly, or instead to shoot for interaction models and GUIs. Query/Browse/Mine.

6.2 Extensible Query Engines

Volcano, Cascades, River systems (Gray & Remzi)

6.3 Stream Support in NW and OS

Click, SCOUT.

6.4 Stream Support in DBMS Research

Mark Sullivan. Hancock. Recent workshop after SIGMOD.

6.5 Interoperability and Standards

APIs and functionality needs. Working with the lowest common denominator.

Structured data, XML, and wrappers.

IETF, etc.

---

5 To decide between these alternatives in an online query processing environment, it would be important to be able to model a user’s preference for updated information as opposed to overall performance – this is captured as an “animation speed” preference in [7].
6.6 Threads and Messages
Lauer and Needham, Scout. Gribble, Larus.

6.7 Enterprise Messaging and Workflow
Including MQ, pub/sub, etc.

7 Status and Future Work

As we hope is clear from this overview, Telegraph is an ambitious system, and much work remains to be done. Currently, Telegraph is a fairly robust adaptive, single site relational query processor, particularly suited to running queries over web data sources, with online query processing controlled from a custom user interface. The primacy of this research thrust is in part the result of our initial application for the 2000 presidential election, and the hard “ship date” that presented. Our work on adaptive, cluster-based parallelism is also working in a state worthy of demonstration, though certain many questions remain in that context. Our agenda for web query work includes further study of online query interfaces and processing, including semi-automatic techniques for query-composition and schema discovery.

However, in a distributed environment we have only initial results, and this is perhaps the environment most apropos for emerging sensor-centric and peer-to-peer applications. P2P future work blah blah.

On the sensor front, we believe that our Fjords infrastructure is a first step in a practical agenda to knit sensors into a rich dataflow architecture. A natural next step is to study how an algebra of typical sequence operators will interact with our adaptivity infrastructure. However, many speculative questions remain. We have yet to really study the possibilities and practicalities of pushing computation into a sensor network, which may prove important for improved bandwidth and power consumption. We have also yet to begin studying the typical data quality and node failure issues that arise in this context, and the effects of these issues on natural operators.