TelegraphCQ: Continuous Dataflow Processing for an Uncertain World[+]

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong*, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman**, Fred Reiss, and Mehul Shah

University of California, Berkeley
*Intel Berkeley Laboratory
**IBM Almaden Research Center
http://telegraph.cs.berkeley.edu

Abstract

Increasingly pervasive networks are leading towards a world where data is constantly in motion. In such a world, conventional techniques for query processing, which were developed under the assumption of a far more static and predictable computational environment, will not be sufficient. Instead, query processors based on adaptive dataflow will be necessary. The Telegraph project has developed a suite of novel technologies for continuously adaptive query processing. The next generation Telegraph system, called TelegraphCQ, is focused on meeting the challenges that arise in handling large streams of continuous queries over high-volume, highly-variable data streams. In this paper, we describe the system architecture and its underlying technology, and report on our ongoing implementation effort, which leverages the PostgreSQL open source code base. We also discuss open issues and our research agenda.

1 INTRODUCTION

The deployment of pervasive communications infrastructure ranging from short-range wireless *ad hoc* sensor networks to globe-spanning intra- and internets has enabled new applications that process, analyze, and react to disparate data in a near real-time manner. Examples include: event-based business processing, profile-based data dissemination, and query processing over streaming data sources such as network monitors, sensors, and mobile devices. Such applications present challenges that cannot be met by existing database and data management technology. These challenges stem from their large scale, their deeply-networked nature, the unpredictability of the environment, and the need for close interaction with users.

In emerging networked environments, data is the commodity of interest, and like any commodity, its value is realized only when it is moved to where it is needed. In contrast to traditional data processing environments where data can be assumed to reside statically in known locations, data in these new applications is constantly moving and changing. This fluidity leads us to view data management for these emerging applications as dataflow processing that must monitor and react to streams of information as they pass through the network.

1.1 Data Movement Implies Adaptivity

Traditional database query processing approaches are inappropriate as a substrate on which to build dataflow processing support for a number of reasons:

Streaming Data – In contrast to traditional database systems where the query processor can operate by "pulling" data from the disk (say, using an iterator model), our target applications involve *streaming data* where the data is continually "pushed" to the query processor. This subtle difference has dramatic implications for the design of the query processor. Most importantly, while a traditional query processor can *orchestrate* the movement and handling of data, a query processor for data streams must instead *react* to arriving data. The arrival rate of the data streams may be extremely high or bursty, thus placing constraints on processing time or memory usage; typically, data must be processed on-the-fly as it arrives and can be spooled to disk only in the background.

Furthermore, while a traditional processor can rely on detailed statistics about data stored by that system, reliable statistics for streaming data are not readily available. Finally, it is important to note that in many data streams, time and/or ordering are inherently important, thus queries over streaming data are likely to be significantly different than queries over traditional data.

Continuous Queries (CQ) – Dataflow processing applications often have a monitoring or filtering aspect in which queries are continuously active. As new data arrives at the query processor, it is routed through the set of active queries. Such continuous query processing turns

[+] This work was funded in part by the NSF under grants IIS-0086057, SI-0122599, and EIA-0207603, by the IBM Faculty Partnership Award program, and by research funds from Intel, Microsoft, and the UC MICRO program.

traditional database system architecture on its head. In a traditional system, the arrival of queries initiates access to a stored collection of data, while here, the arrival of data initiates access to a stored collection of queries. As would be expected, this inversion dictates a very different approach to query processor architecture.

Another important aspect of continuous queries is that the streams over which they are executed can be effectively infinite. Queries over infinite streams require different query semantics, non-blocking query operators, and new support for fault tolerance. Operators must continuously return incremental results and may need to be defined to process subsets (or windows) of the input. Furthermore, continuous queries can be extremely long-lived, so they are susceptible to changes over time to performance and load, data arrival rates, or data characteristics. Care must be taken to reduce the amount of state such queries accumulate, and this state must be preserved and possibly migrated for fault tolerance and load balancing.

Shared Processing - The combination of long-running continuous queries and on-the-fly processing of data streams necessitates new mechanisms for sharing query processing work. In order to avoid blocking and having to interrupt the dataflow, data should be processed simultaneously by all queries that require it as it flows through the system. Processing each query individually can be slow and wasteful of resources, as the queries are likely to have some commonality. Thus, shared processing must be a fundamental capability of the system. Furthermore, this shared processing must be made robust to the addition of new queries and the removal of old ones over time, so on-the-fly adaptivity must be an essential component of any solution for shared processing of continuous queries.

Other Sources of Unpredictability – Finally, there are a number of other features of our target applications that render existing static query processing techniques inappropriate. First, as mentioned above, deeply networked environments can be highly volatile due to the number of communications links and disparate systems and devices involved, data loss (particularly for sensor networks), and uncertainty due to the unavailability of load information, statistics, and cost estimates about remote systems or remotely stored or sensed data. Second, dataflow processing is often part of a larger control loop in which query results may be used to affect the environment or redirect further query processing or data production. In many cases, users will be directly interacting with the system while results stream out. Users may choose to modify their queries on the basis of previously returned information or other factors. The system must be able to gracefully adjust in response to user needs.

For all of the reasons outlined above, we have developed a new architecture for shared, continuous, dataflow processing. Our approach is distinguished from other projects on data stream query processing (e.g., [CDTW00,CCCC+02, BBDM+02]) by our emphasis on adaptability. That is, we have designed the data management components of our system to be able to quickly evolve and adjust to radical changes in data availability and content, systems and network characteristics, and user needs and context. These considerations have served as the guiding principles underlying the design of TelegraphCQ.

1.2 TelegraphCQ - Background

The Telegraph project at UC Berkeley began in early 2000 with the goal of developing an Adaptive Dataflow Architecture for supporting a wide variety of data-intensive, networked applications. The Telegraph concept grew out of earlier projects on adaptive relational query processing aimed at building systems that could adjust their processing on the fly, in response to changes in user needs [HACO+99] or to intermittent delays in accessing data across wide-area networks [UFA98].

The basic technologies underlying Telegraph were developed to provide adaptivity to individual dataflow graphs. The first version of Telegraph [SMFH01] was deployed to support Federated Facts and Figures (FFF), a query system for deep-web data. FFF made no attempt to exploit commonality among concurrently active queries. Instead, it focused on a single-user scenario, providing efficient early "partial" results to queries with interactive user control [RH02].

Recently, we have built two prototypes extending Telegraph to support shared processing over streams, namely CACQ [MSHR02], and PSoup [CF02]. These prototypes demonstrated substantial advantages of our adaptive framework for shared processing over streams and showed how the adaptivity framework could be extended to incorporate such sharing. Both systems, however, had significant limitations. In particular, these systems: 1) restricted their processing to data that could fit in memory, 2) did not investigate scheduling and resource management issues for queries with little or no overlap, 3) did not explicitly deal with the notion of Quality of Service (QoS) for adapting to resource limitations, and 4) did not explore opportunities for varying the degree of adaptivity to tradeoff flexibility and overhead.

Building on these initial prototypes, we have embarked on a complete redesign and reimplementation of our system, with a focus on support for shared, continuous query processing over query and data streams. We refer to this system as TelegraphCQ to distinguish it from the Telegraph project's broader focus on adaptive dataflow in general, and to emphasize the challenges we are addressing in our new implementation.

In the remainder of this paper we describe our ongoing design of TelegraphCQ, focusing on how it addresses the challenges outlined above. We begin by reviewing the basic adaptive mechanisms in Telegraph.

2 ADAPTIVE BUILDING BLOCKS

The main components of Telegraph are shown in Figure 1. Telegraph consists of an extensible set of composable dataflow *modules* or operators that produce and consume records in a manner analogous to the operators used in traditional database query engines, or the modules used in composable network routers [KMCJ+00]. The modules can be composed into multi-step dataflows, exchanging records via an API called *Fjords* [MF02] that can support communication via either "push" (asynchronous) or "pull" (synchronous) modalities. Dataflows are initiated by clients either via an ad hoc query language (a basic version of SQL), or via a scripting language for representing dataflow graphs explicitly. Telegraph maintains a metadata catalog of data ingress "wrappers" or "gateways" that provide access to a variety of data sources including remote web and peer-to-peer sources on the Internet, local caches and files, and live sensor networks. Client communication to Telegraph can be done via TCP/IP sockets (e.g. from an applet running in a remote browser), or via local command-line interfaces.

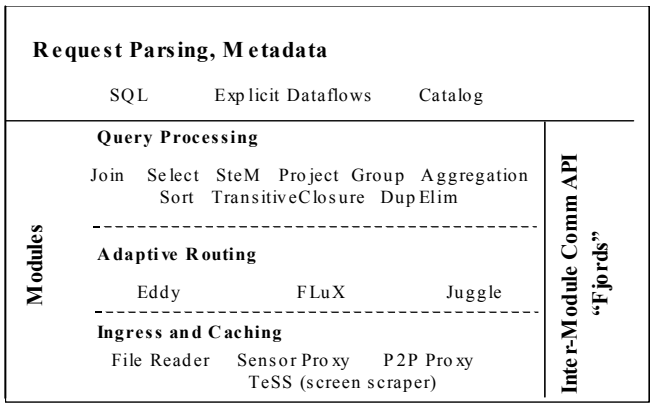

Figure 1 - Telegraph Architecture

2.1 Module Types

As shown in Figure 1, Telegraph contains three types of modules:

Ingress and Caching – These modules are responsible for interfacing with external data sources. Most Ingress modules are fairly traditional wrappers, such as an HTML/XML screen scraper (called "TeSS", the Telegraph Screen Scraper), a proxy for fetching data from popular peer-to-peer networks (called "TeleNap"), and a local file reader; these modules are akin to read-only access methods in a relational database, but reach out to remote data sources rather than local disks or indexes. Such modules may also cache data locally to hide network delays. In addition more sophisticated Ingress modules can be built that can also send messages back to the network. For example a sensor proxy may send control messages to adjust the sample rate of a sensor network based on the queries that are currently being processed [MF02]. Similarly, the TeSS module is able to pass bindings into remote websites to perform lookups.

Query Processing – In Telegraph, query processing is performed by routing tuples through query modules. These modules are pipelined, non-blocking versions of standard relational operators such as joins, selections, projections, grouping and aggregation, and duplicate elimination. In addition, Telegraph uses a special type of module known as a State Module (SteM) [RDH02]. SteMs are described in Section 2.2.

Adaptive Routing - Telegraph does not rely upon a traditional query plan, but instead, constructs a query plan that contains adaptive routing modules, which are able to "re-optimize" the plan on a continuous basis while a query is running. Eddies are modules that adaptively decide how to route data to other query operators on a tuple-by-tuple basis [AH00], choosing orderings among commutative modules. Juggle performs online reordering for prioritizing records by content [RRH99]. Flux routes tuples among machines in a cluster to support parallelism with load-balancing and fault-tolerance [SCHF03]. Architecturally, these modules are indistinguishable from the other more traditional modules: they simply consume and produce records via the usual Fjords API. However, these modules can serve all the roles traditionally handled by an offline query optimizer: ordering of operations, choice of access and query modules, and partitioning/replication of dataflows across multiple machines. Moreover, these modules can reconsider and revise these decisions while a query is in flight.

2.2 Adaptive Processing W/Eddies & SteMs

Of particular relevance to the development of TelegraphCQ is the flexibility that is obtained by combining the Eddy adaptive tuple routing module with State Modules (SteMs). The role of an Eddy is to continuously route tuples among a set of other modules according to a *routing policy*. Eddies are intended to support a partially or completely commutative set of modules, whose inputs and outputs are connected to the Eddy. This topology allows the Eddy to intercept tuples that flow into and out of these modules, observing the module behavior and choosing the order that tuples take through the modules. When one of the modules processes a tuple *t*, it can generate other tuples (e.g., by concatenating the input tuple with other tuples) and send them back to the Eddy for further routing. A module can also optionally return (or *bounce back*) *t* to the Eddy if *t* requires additional processing. A tuple is sent to the Eddy's output if all the modules connected to the Eddy have successfully handled it. The Eddy shuts down its connected modules when the end of all of its input streams (or base tables) has been reached, and each module has finished processing all of the tuples sent to it.

In order to enable tuples to be routed individually, each tuple must have some additional state with which it is associated. The exact structure and location of this state

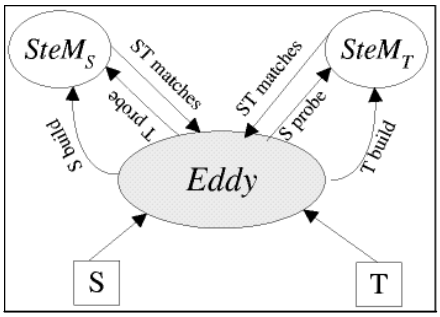

Figure 2 - Eddy and SteMs

depends on the routing policy and implementation, but at a minimum, for an Eddy representing a single query, the state must indicate the set of connected modules successfully visited by the tuple. Note that this state may be attached to each tuple as in [AH00], or similar tuples may be logically batched together and associated with common state information stored by the Eddy.

A SteM is a temporary repository of tuples, essentially corresponding to half of a traditional join operator. It stores homogeneous tuples (i.e., tuples spanning the same set of tables) formed during query processing and supports insert (*build*), search (*probe*), and optionally delete (*eviction*) operations. Two kinds of tuples can be routed to a SteM. When a tuple $t \in T$ (a *build tuple*) is routed to $SteM_T$, t is added to the set of tuples in $SteM_T$. When a tuple $p \notin T$ (a *probe tuple*) is routed to $SteM_T$, $SteM_T$ returns *concatenated matches* for it to the Eddy. These concatenated matches are the tuples in $\{p\}$ *join* $SteM_T$ that satisfy all query predicates that can be evaluated on the columns in p and T.

In order to speed processing, SteMs can be augmented with indexes. For example, Figure 2 shows an example of using an Eddy and two SteMs to implement a symmetric hash join between two relations (S and T). In the join shown in Figure 2, a hash index would be built on the join attributes in each SteM. When an S tuple arrives, it is first sent as a build tuple to $SteM_S$ and then sent as a probe tuple to $SteM_T$. *ST* matches produced from either SteM are routed to the output. This routing, combined with hash indexes on the two SteMs implements an adaptive symmetric hash join. This approach can be naturally extended to provide N-way symmetric joins.

As a different example of the use of SteMs, consider a join in which table S is joined with a remote index on table T (e.g. T is a web lookup form wrapped by TeSS). The best way to implement index joins with remote sources is in an asynchronous fashion as described in [GW00], requiring a SteM on S (a *rendezvous buffer*) to hold S tuples pending matches from the index. In order to minimize latency, a SteM on T should also be built, as a cache of previous expensive T lookups, as in [HN96]. This dataflow looks almost identical to Figure 2, except that S probes are also routed from the Eddy directly to an index access method on T.

The two plans described above can be combined into a single nearly identical plan that contains one Eddy, two SteMs, and *both* access methods on T. In that case, the Eddy can essentially run both query plans at the same time, by routing the tuples in different ways but sharing the work of building the SteMs. In some sense the Eddy is doing online competitive optimization in the spirit of [Anto93], but (a) it considers both the choice of access methods and join algorithms, (b) it can change its mind multiple times during the run of the query, and (c) the tuples accessed by one plan are reused by the other, so there is minimal wasted effort. Described differently, the Eddy and SteMs dynamically design a *hybrid* join algorithm. Eddies and SteMs can also dynamically control the other decisions in query optimization, including module ordering, and choice of query spanning tree (choosing which pairs of relations to join). The benefits of query processing with Eddies and Stems are addressed in more detail in Raman et al. [RDH02], which also includes experiments showing the performance benefits of join hybridization.

It is important to note that any number and combination of modules can be connected to an Eddy – including of course, other Eddies. Each individual Eddy provides a scope for adaptivity; modules at the input or output of an Eddy are not considered in the Eddy's adaptive decision-making, and thus, do not contribute to the overhead thereof.

2.3 Fjords – InterModule Communication

The glue that binds the various modules together to form a query plan is an inter-module communications API that we call Fjords. The key advantage of Fjords is that they allow query plans to use a mixture of *push* and *pull* connections between modules, thereby being able to execute query plans over any combination of streaming and static data sources. The Fjord API is designed so that the modules themselves can be written in a manner that is agnostic as to whether their inputs and outputs are all pushed, all pulled, or some combination of the two.

The insight behind Fjords is that in interactive, adaptive, and streaming systems, the traditional iterator model breaks down because the query processor cannot afford to block waiting for long running modules to complete, for slow web pages to return results, or for individual sensors, which may have run out of power or temporarily disconnected.

One way to deal with such problems is to interpose "Exchange" modules [Graf93] between the producers and consumers in a query plan. With Exchange, a producer running in its own thread (or on another machine) delivers results to the Exchange module, which queues them and synchronously delivers them to the consumer when they are needed. Using Exchange, however, the consumer is still forced to block if no data is available, due to the iterator model. Such blocking can limit adaptivity.

Instead, Fjords allow pairs of modules to be connected by various types of queues. For example, a pull-queue is implemented using a blocking dequeue on the consumer side and a blocking enqueue on the producer side. A push-queue is implemented using non-blocking enqueue and dequeue; control is returned to the consumer when the queue is empty. The non-blocking dequeue allows the consumer to pursue other computation or yield the processor when no data is available. Of course, if desired, Fjords can provide Exchange semantics using a blocking dequeue and a non-blocking enqueue.

2.4 Flux: Scaling Up Dataflow Processing

Scalability is a key concern for dataflow processing systems. The traditional approach for scaling a query is to horizontally partition its constituent operators across a shared nothing cluster and use dataflow processing to execute it [DG92]. In volatile environments, such as the ones for which Telegraph is intended, the optimal partitioning of the internal state and input streams of a dataflow's constituent operators is likely to change over time. Thus, to execute efficiently, operators must periodically adjust their partitioning midstream, while still executing. For operators with large, ever-changing internal state, online repartitioning is especially difficult and costly. Moreover, in a shared-nothing environment, machines are likely to experience faults causing portions of continually executing dataflows to lose accumulated operator state and in-flight data. To deal with these volatilities, we introduce a module called Flux[1].

Flux is a generalization of the Exchange module [Graf93] and like an Exchange, is an opaque dataflow module interposed between a producer-consumer operator pair in a pipelined, partitioned dataflow. In addition to the data partitioning and routing functions of the Exchange, Flux provides two additional features: load balancing and fault tolerance. Load balancing is provided via online repartitioning of the input stream and the corresponding internal state of operators on the consumer side. The Flux state movement protocol employs buffering and reordering mechanisms to smoothly repartition operator state across machines with minimal impact to ongoing processing.

Flux provides fault-tolerance for dataflows by leveraging these state movement mechanisms to replicate an operator's internal state and in-flight data. For critical dataflows that require high-availability, Flux provides a loosely coupled process-pair-like mechanism for quick failover. On failure, Flux automatically recovers lost in-flight data and operator state on the remaining non-faulty machines and continues processing without human intervention. The online repartitioning mechanisms then take over to provide efficient rebalancing of execution.

In addition, Flux can be parameterized to provide varying degrees of replication at different levels in a dataflow and on per-partition basis for modules partitioned across a cluster. This flexibility allows unneeded reliability to be traded for improved performance. In essence, Flux exposes a reliability-based quality-of-service "knob" for dataflows. By inserting Flux at appropriate points in a dataflow, a designer can build a dataflow that degrades in a controlled fashion in the face of resource imbalances and machine faults across a shared-nothing platform.

[1] Flux: Fault-tolerant, Load-balancing eXchange

3 INITIAL CQ APPROACHES

Having presented the philosophy and core mechanisms of the original Telegraph system, we now describe our early work on extending them to support shared, continuous query processing. There are two main components of this work: *CACQ*, an extension of the Eddy and SteMs mechanisms to support multiple continuous queries [MSHR02] and *PSoup*, a further extension to CACQ that supports access to previously-arrived data and intermittent connectivity [CF02]. Both of these schemes were implemented as relatively natural extensions to the initial Telegraph implementation described in Section 2.

3.1 CACQ

CACQ was the first continuous query engine to exploit the adaptive query processing framework of Telegraph. The key innovation in CACQ is the modification of Eddies to execute multiple queries simultaneously. This is accomplished by essentially having the Eddy execute a single "super"-query corresponding to the disjunction of all the individual queries posed by the clients of the system. Extra state, called *tuple lineage*, is maintained with each tuple as it passes through the CACQ process, to help determine the clients to which the output of the disjunctive CACQ query should be transmitted.

Another key feature of CACQ is its use of *grouped filters* to optimize selections in the shared execution of the individual queries. A grouped filter is an index for single-variable boolean factors over the same attribute. When a new query is inserted into the system, it is decomposed into its individual boolean factors. The single-variable boolean factors are then inserted into appropriate grouped filters. Multi-variable boolean factors are inserted into SteMs. The Eddy subsequently routes a tuple that enters the system through all the grouped filters and SteMs that are *interested* in it.

The details of the tuple lineage, the grouped filters and the execution of joins using SteMs is described in Madden et al. [MSHR02]. Performance experiments reported in that paper indicate that due to its adaptive nature, the CACQ system is able to match or significantly exceed the performance of existing static continuous query systems under a variety of workloads.

3.2 PSoup

PSoup extends the mechanisms developed in CACQ in two main ways: 1) it allows queries to access historical data and 2) it adds support for disconnected operation –

users can register queries with the system and return intermittently to retrieve the latest answers.

The key innovation in PSoup is that it *treats data and queries symmetrically*, thereby allowing new queries to be applied to old data and new data to be applied to old queries. It does this by indexing queries into a *query SteM*, which can be thought of as a generalization of the notion of a grouped filter. PSoup also supports intermittent connectivity by separating the computation of query results from the delivery of those results. PSoup continuously computes the answers to all active queries, effectively materializing the results until they are specifically requested.

As shown in Figure 3, the essential idea behind PSoup's execution model for such queries is to treat query processing as a symmetric join between data and queries. When a client first registers a query, The SELECT-FROM-Where clause of the query is extracted and inserted into a Query SteM, and is then applied to previously arrived data stored in Data SteMs. This application of "new" queries to "old" data is how PSoup executes queries over historical data. Similarly, when a new data element arrives, it is inserted into the appropriate Data SteM, and is then applied to previously specified queries stored in the Query SteM. This act of applying "new" data to "old" queries is how PSoup supports continuous queries.

PSoup continually runs the data/query join, *materializing* the results in a special Results Structure. Queries in PSoup contain a time-based window specification. When a previously registered query is invoked, the window is imposed on the Results Structure to retrieve the current results. The materialization of results is the key to supporting disconnected operation and also enables efficient support for set-based queries. The performance study presented in [CF02] shows the benefits of the PSoup materialization strategy.

4 TELEGRAPHCQ

The Telegraph implementation and extensions that we have built to date have enabled us to explore novel implementations for adaptive CQ processing mechanisms, and showed significant advantages over more traditional approaches in a range of application scenarios. However, as discussed in Section 1.3, these prototypes had a number of limitations that we are addressing in the development of the TelegraphCQ system. Specifically, we are designing TelegraphCQ with a focus on the following issues: 1) scheduling and resource management for groups of queries, 2) support for out-of-core data, 3) variable adaptivity, 4) dynamic QoS support, and 5) parallel cluster-based processing and distribution.

In this section we first present an overview of the window-based query semantics to be supported by TelegraphCQ. We then describe the design of the system, focusing on how we are leveraging the PostgreSQL code base. Finally, we discuss some of the open issues we are currently addressing in the design.

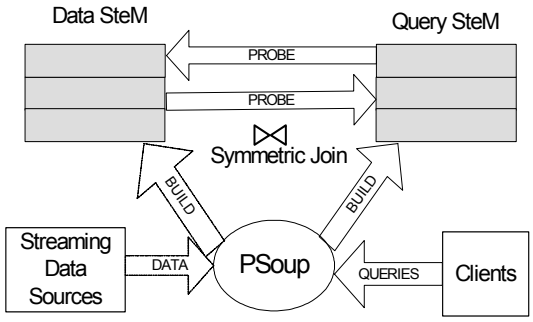

Figure 3 - PSoup

4.1 Window Semantics in TelegraphCQ

TelegraphCQ supports continuous queries over a combination of tables and data streams. To deal with data streams whose length is unbounded, certain operations, such as joins, can only be run over finite windows on these streams. In order to support a variety of query types, TelegraphCQ supports rich windowing schemes over both the portion of the stream that has already arrived, as well as those portions that will arrive in the future. TelegraphCQ also provides flexible mechanisms for delivering the query results generated over these windows. The engine allows the results of the execution of the query over consecutive windows to either be pushed out to the user as in CACQ, or to be pulled by the user upon demand, as in PSoup. In this section we describe the windowing functionality provided by TelegraphCQ. We also briefly discuss the impact of this functionality on our design.

4.1.1 Windows for Input Streams

Two popular windowing schemes in the context of stream query processing are *landmark* and *sliding* windows [GKS01]. For landmark queries, the *older* end of the window is fixed, while the *newer* end of the window moves forward with the arrival of new tuples in the stream. In contrast, for sliding window queries, both the ends move forward in unison with the arrival of new tuples. The semantics offered by landmark and sliding windows, however, only cover a small fraction of the interesting applications over data streams.

For example, *landmark* and *sliding* windows do not capture the semantics of a query executed upon the arrival of every *n* tuples, nor can they describe windows occur in the past. As another example, consider a *browsing* system where the user might want to query historical portions of the stream using windows that move *backwards* starting from the present time. Traditional landmark and sliding windows cannot be used in such applications.

The semantics of queries in TelegraphCQ are as follows. For every instant in time, a window on a stream defines a set of tuples over which the query is to be executed. Since each execution of the query produces a set, the output of a query is presented to the end-user as a sequence of sets, each set being associated with an instant

in time. TelegraphCQ allows multiple simultaneous notions of time, such as logical sequence numbers or physical time. In order to accommodate loosely synchronized distributed data sources, we treat time as a partial order, rather than as a complete order. We have designed an algebra that extends the standard relational operators to operate on streams and to allow a stream defined using one notion of time to be transformed into a stream using another.

We support much more general windows than the landmark and sliding windows described above. This is done using a for-loop construct to *declare* the sequence of windows over which the user desires the answers to the query: a variable "t" moves over the timeline as the for-loop iterates, and the left and right ends (inclusive) of each window in the sequence, and the stopping condition for the query can be defined with respect to this variable "t".

The for-loop contains a *WindowIs* statement for each stream in the query: an input without a corresponding WindowIs statement is assumed to be a static table by default. There is one for-loop for every group of streams that exhibit the same window transition behavior[2]. Note that our notion of a for-loop is intended as a powerful, *low-level* mechanism rather than a *user-level* query language construct. The syntax of the for-loop is as follows:

```
for(t=initial_value; continue_condition(t); change(t)){
    WindowIs(Stream A, left_end(t), right_end(t));
    WindowIs(Stream B, left_end(t), right_end(t));
    …
}
```

We now demonstrate the functionality of the window mechanism using several examples. All the queries in these examples use the following schema for the daily closing prices of stocks:

```
ClosingStockPrices(
    long timestamp;
    char(4) stockSymbol;
    float closingPrice;)
```

We assume that the stream starts with logical timestamp 1. There is one entry for every trading day for every stock symbol. For simplicity, we assume that Microsoft (MSFT) has been trading since the beginning of the stream.

1. Snapshot query: These queries execute exactly once over one window. Example: "*Select the closing prices for MSFT on the first five days of trading".*

```
SELECT closingPrice, timestamp
FROM ClosingStockPrices
WHERE stockSymbol = 'MSFT'
for (; t==0; t = -1 ){
    WindowIs(ClosingStockPrices, 1, 5);
}
```

[2] The transition behavior of windows is determined by the units used to define the windows, and the increment statement and continuation condition in the for-loop.

2. Landmark query: The input windows of these queries have a fixed beginning point in the timeline, and a forward moving endpoint. Example: "*Select all the days after the hundredth trading day, on which the closing price of MSFT has been greater than $50. Keep this query standing in the system for a thousand trading days".*

```
SELECT closingPrice, timestamp
FROM ClosingStockPrices
WHERE stockSymbol = 'MSFT' and
    closingPrice > 50.00
for (t = 101; t <= 1000; t++ ){
   WindowIs(ClosingStockPrices, 101, t);
}
```

3. Sliding query: The input windows of these queries have forward moving beginning and end points. Example: *"On every fifth trading day starting today, calculate the average closing price of MSFT for the five most recent trading days. Keep the query standing for fifty trading days".* (note: ST = the start time of the query.)

```
Select AVG(closingPrice)
From ClosingStockPrices
Where stockSymbol = 'MSFT'
for (t = ST; t < ST + 50; t +=5 ){
   WindowIs(ClosingStockPrices, t - 4, t);
}
```

Notice that this window *hops*, rather than moving smoothly over the timeline. Windows can also be defined to move on-demand, or in the reverse-timestamp direction by appropriately setting the increment statement for "t" in the for-loop.

4. Temporal Band-Join: These queries join tuples in one stream with tuples in another based on timestamp. Example: "*For the five most recent trading days starting today, select all stocks that closed higher than MSFT on a given day. Keep the query standing for twenty trading days".*

```
Select c2.*
FROM ClosingStockPrices as c1,
      ClosingStockPrices as c2
WHERE c1.stockSymbol = 'MSFT' and
    c2.stockSymbol!= 'MSFT' and
    c2.closingPrice > c1.closingPrice and
    c2.timestamp = c1.timestamp
for (t = ST; t < ST +20 ; t++ ){
   WindowIs(c1, t - 4, t);
   WindowIs(c2, t - 4, t);
}
```

4.1.2 Effect of Window Semantics on System Design

The different types of windows can impose significantly different requirements on the design of the query processor and its underlying storage manager. One fundamental issue has to do with the use of logical (i.e., tuple sequence number) vs. physical (i.e., wall clock) timestamps. If the former is used, then the memory requirements of a window can be known *a priori,* while in

the latter case, memory requirements will depend on fluctuations in the data arrival rate.

Another issue related to memory requirements has to do with the type of window used in a query. Consider the execution of a MAX aggregate over a stream. For a landmark window, it is possible to compute the answer iteratively by simply comparing the current maximum to the newest element as the window expands. On the other hand, for a sliding window, computing the maximum requires the maintenance of the entire window.

Finally, the direction of movement, and the "hop" size of the windows (the distance between consecutive windows defined by the for loop) also have significant impact on query execution. For instance, if the hop size of the window exceeds the size of the window itself, then some portions of the stream are never involved in the processing of the query.

4.2 TelegraphCQ Design Overview

Having presented the notion of windowed queries that TelegraphCQ supports, we can now describe our ongoing implementation. In this section we outline the software architecture of TelegraphCQ focusing first on how we are adapting the architecture of PostgreSQL to enable shared processing of continuous queries over streaming sources. We then describe the new components that comprise TelegraphCQ.

4.2.1 Approach

After considerable analysis (and some hand-wringing) we decided throw out our existing Java-based prototypes and to implement a completely new system using C/C++. While there are many considerations in choosing between Java and C/C++ for a systems development project, in this case, the over-riding factor was our decision to heavily leverage the open source PostgreSQL code base. Although TelegraphCQ is quite different from a traditional query processor, there is a fair amount of code surrounding the main query processing modules that we can profitably reuse.

Figure 4 shows the basic process structure of PostgreSQL. PostgreSQL uses a process-per-connection model. Data structures shared by multiple processes, such as the buffer pool, latches, etc. are located in shared memory. A *Postmaster* forks new server processes in response to new client connections. Within a server process, a *Listener* is responsible for accepting requests on a connection and returning data to the client. When a new query arrives it is parsed, optimized, and compiled into an access plan that is then processed by the query *Executor*.

The components we can use with only minimal change in TelegraphCQ are shaded in dark gray in Figure 4. These include: the Postmaster, Listener, System Catalog, Query Parser and Optimizer. Components shown in light gray (the Executor, Buffer Manager and Access Methods) are pieces we expect to leverage but only with significant changes. In addition, by adopting the front-end

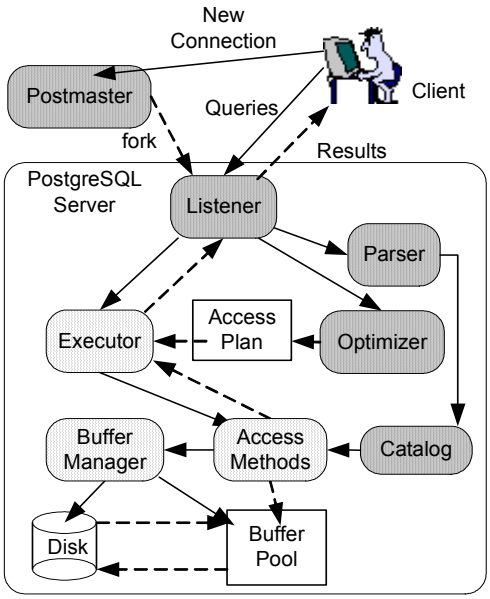

Figure 4 - Architecture of PostgreSQL

components of PostgreSQL we also get to access to important client-side call-level interface implementations (not shown in the figure) such as ODBC and JDBC.

Our chief challenge in using PostgreSQL is supporting the TelegraphCQ features it was not designed for: streaming data, continuous queries, shared processing and adaptivity. Another major issue is that PostgreSQL's process-per-connection model implementation is not thread-safe, while for performance reasons, multi-threading is an important feature of the TelegraphCQ design. We are thus taking a pragmatic approach to the implementation: we use the existing process model when we reuse old code and venture into multi-threading only with exclusively new code.

Figure 5 shows how we are adding the TelegraphCQ functionality to the PostgreSQL code base. The figure shows (as ovals) the three processes that comprise the TelegraphCQ server. These processes are connected using a shared memory infrastructure. The rightmost process in the picture is the "FrontEnd", which contains the Listener, Catalog, Parser and Optimizer. The actual query processing takes place in a separate process called the "Executor". Finally, a "Wrapper" process is used to host the data ingress operators.

As in PostgreSQL, the Postmaster listens on a well-known port; it forks a FrontEnd process for each fresh connection it receives. Since each connection can have multiple open cursors, we use a proxy service (shown on the right of Figure 5) to collect individual requests from clients and instantiate multiple cursors using a single connection. If necessary, multiple proxies can be used to overcome limitations on the number of permitted open cursors for any connection.

The listener accepts multiple continuous queries and adds them dynamically to the running executor. When a query is received, the server parses, analyzes, and

optimizes it into an *adaptive* plan, that is, a plan that includes the adaptive operators described in Section 2. The plans are then placed in the query plan queue (QPQueue), in a shared memory segment, for the executor. The executor continually picks up fresh queries from the shared memory segment. These plans are dynamically folded into the running queries in the executor. Query results are placed in client-specific output queues, which are also located in shared memory segments. The listener picks up results from the output queues and sends them to the client proxy for distribution to the clients.

4.2.2 The TelegraphCQ Executor

A key challenge in designing the new executor is the mapping of our shared continuous processing model onto a thread structure that will allow for adaptivity while incurring minimal overhead. In theory, a single Eddy running in a single thread could be used to run all the queries in the system (as in CACQ and PSoup), including those involving totally unrelated streams. Such an approach has a number of problems, however, as the Eddy mechanism was not intended to be a generalized scheduler. For example, it is not tailored to enforce policies for resource management across disjoint classes of queries. We expect to support large numbers of active, standing queries, so we need to avoid the overhead associated with making each query a separate thread yet we need multiple threads in order to exploit SMP and cluster parallelism.

As a result, the TelegraphCQ executor is being developed using a multi-threaded approach in which the threads provide execution context for multiple queries encoded using a non-preemptive, state machine-based programming model. We use the term "Execution Object" (EO) to describe the threads of control in the TelegraphCQ executor. Each EO is mapped to a single system thread. (Note that Figure 5 shows a system with a single EO instantiated.) An EO consists of a scheduler, one or more event queues, and a set of non-preemptive Dispatch Units (DUs) that can be executed based on some scheduling policy. Unlike EOs, which are visible to the operating system, DUs are merely abstractions that represent entities that perform "work" in the system. DUs are responsible for maintaining their own state. DUs are non-preemptive, but they follow the Fjords model described in Section 2.3, which gives us control over their scheduling.

A DU can be run in one of the following modes:

1. A single "traditional" PostgreSQL query plan with the standard query executor.
2. A single-Eddy query plan with Fjord-style operators.
3. A shared "continuous query" mode with an Eddy and Fjord-style operators.

Like PostgreSQL, TelegraphCQ uses surrogate objects to represent tuples during query processing. While running "traditional" plans TelegraphCQ uses the PostgreSQL format for surrogates. While running in the context of an Eddy, however, due to the continuously changing join order, the intermediate tuples can be in a multitude of formats. In addition, they must carry extra information such as bitmaps for CACQ. Thus, an enhanced surrogate object format is used to represent intermediate tuples in the Eddy-based modes.

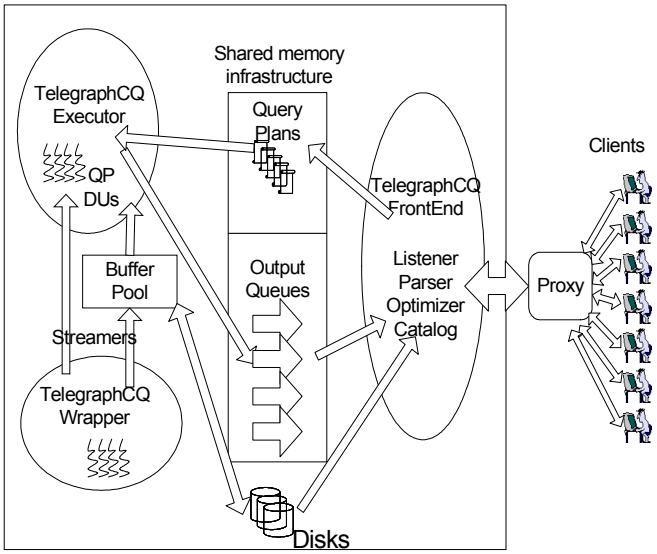

Figure 5 – TelegraphCQ Architecture

A key design decision in the executor is how to map queries onto the model of pre-emptively scheduled EO system threads containing non-preemptive DUs. The goal is to separate queries into classes that have significant potential for sharing work. This determination is made based on the set of streams and tables over which the queries are defined, which we call the query *footprint*. In the current implementation, we create query classes for disjoint sets of footprints. However, we intend to investigate more sophisticated sharing schemes as well as techniques for maintaining and adjusting the classes as queries enter and leave the system.

4.2.3 Ingress Operators

The final aspect of the system we discuss here is the Wrapper mechanism that allows data to be streamed into the system. By wrapping streams, newly arriving streamed data can be accessed using mechanisms similar to those used for previously arrived or even static data. However, an overarching principle of TelegraphCQ is to avoid blocking operations, save accesses to disk. For this reason, wrappers in TelegraphCQ are placed in a *separate* process, where they can be accessed in a non-blocking manner (a la Fjords). Two types of sources are supported:

1. Pull sources, as found in "traditional" federated database systems.

2. Push sources, where connections can be initiated either by the Wrapper (Push-client) or by the data source itself (Push-server).

With pull and push-client sources the Wrapper connects to the source. In contrast, push-server sources connect to a well-known port served by the Wrapper process. In either case, the responsibility of fetching data from the network devolves to the Wrapper process, which uses a pool of threads to implement non-blocking I/O from the network.

Streamed data is delivered from the Wrapper process to the Executor via *streamers*. A streamer produces tuples for a *stream*, by preparing them for materialization in the buffer pool (and possibly to disk) or for direct delivery to the Executor. Tuples in buffer pool pages are accessed via a "scanner" operator, which is similar to the standard scan operators in classic systems, except that it is driven by window descriptors. Processing over streamed data that has been (partially) spooled to disk is an area of on-going design, as discussed in the next Section.

4.3 Discussion and Open Issues

In the previous sections we have outlined our ongoing implementation of shared continuous query processing over streams leveraging an existing, traditional database engine. While the basic concepts of our approach have been defined, and the core operators on which the system rests have been developed, there are a large number of important open design questions that we are addressing as part of the TelegraphCQ effort. In this section we briefly discuss some of these issues.

Query Grouping and Sharing. As mentioned in Section 4.2.2, the TelegraphCQ executor partitions queries into execution objects so that queries in the same EO tend to have a high degree of overlap. This approach allows many logical operations to share a few physical SteMs and filters. An open issue is determining how much overlap is required to group a new query into an existing execution object.

Even given a policy for partitioning queries, however, questions about the best policies for routing tuples between operators in a single EO remain. In CACQ, a simple extension of the ticket-based policy presented in [AH00] was shown to provide reasonable performance for some workloads. It remains to be demonstrated that such "best effort" techniques can provide adequate performance for a large class of queries. Furthermore, our approaches to date have been optimized for global throughput, and provide no mechanism for prioritizing queries or preventing a single very expensive query from starving others. Thus, there are several significant open problems with respect to the complexity and quality of routing policies: understanding how ticket based schemes perform under a variety of workloads, and how they compare to (NP-hard) optimal schedule computations; modifying such schemes to adjust the priority of individual queries; and evaluating the feasibility (in terms of computational complexity and quality) of more sophisticated schemes.

Adapting Adaptivity. Our adaptive mechanisms are designed to perform well in environments where little or no cost information is available, or where estimates of such information are unreliable in the long term. As such, they make both *per-tuple* and *per-operator* routing decisions. Such fine-granularity scheduling, of course, does come at some cost -- indeed, we observed scenarios in our previous Java-based implementation where routing decisions could consume significant portions of overall execution time. For this reason, we believe two techniques will play a key role in TelegraphCQ: *batching tuples,* by dynamically adjusting the frequency of routing decisions in order to reduce per-tuple costs; and *fixing operators,* by adapting the number and order of operators scheduled with each decision to reduce per operator costs.

These adjustments constitute a pair of knobs that can be turned as observations of rate of change and relative selectivity vary: when change is slow, or selectivity constant, many tuples should be routed to large, fixed sequences of operators; when change is fast, or selectivities vary wildly, small groups of tuples should be routed to individually scheduled operators. Thus, these knobs serve as the primary mechanism for adapting the adaptivity of TelegraphCQ; implementing them requires investigation into the proper mechanisms for batching and fixing, as well as policies for *automatically* turning knobs based on rates of change and relative selectivity.

Disk-based issues and QoS. Several interesting issues arise when considering disk-based storage for streaming applications. The first issue concerning the design of a storage manager is the technique used to *stream* remote data from diverse push and pull-based sources into the disk and to the executor through the buffer pool. The buffer pool manager must be tuned to both accept new bursty streaming data, as well as service queries that access historical data. The buffer pool must use replacement and eviction policies that can satisfy the multiple simultaneous requests issuing from the shared query processor.

In addition, in scenarios with huge numbers of queries with periodically active windows, the Query SteMs (in addition to Data SteMs) may need to be flushed to disk. In this case, the periodic nature of the windows provides knowledge that can be exploited for prefetching queries from the disk. The streaming nature of the data coupled with the types of queries we describe in Section 4.2.1 raise interesting questions concerning the design of access methods that are best suited for different kinds of windows (backwards moving windows, hopping windows, sliding windows etc.).

Another issue is how to implement the underlying file system. A log-structured file system would enhance write

performance, but for windowed queries of the type presented in Section 4.1, the read workload on the disk resembles that of periodic data broadcasting systems [AAFZ95], which require very different data layout. We are currently designing a storage subsystem that exploits the sequential write workload, while also providing broadcast-disk style read behavior. This effort includes an investigation of the effects of different Eddy routing policies on disk-access behavior.

Queries accessing data that spans memory and disk also raise significant Quality of Service issues, in terms of deciding what work to drop when the system is in danger of falling behind the incoming data stream. Our earlier work on the Juggle operator [RRH99] and on dynamic pipeline processing [UF02] provide mechanisms for pushing user preferences down into the query execution process. Such techniques will need to be integrated into TelegraphCQ.

Egress Modules. Analogous to our *ingress* modules, we also plan to investigate mechanisms for managing and delivering results, which will be encapsulated in *egress* operators. These operators are responsible for handling results of the query execution engine to accommodate different modalities of client interaction. For example, push-based egress operators support interaction where clients are continually streamed query results, while pull-based egress operators may log data and support intermittent retrieval of results. Such operators can encapsulate fault-tolerance mechanisms to support mobile clients that periodically become disconnected, and may encapsulate transcoding services for clients with different capabilities. Most importantly, to efficiently support result delivery to large numbers of clients, we will need operators that provide aggregation and buffering services that interface better with external overlay delivery networks.

Cluster and Distributed Implementations. We are currently extending the Flux module to serve as the basis of the cluster-based implementation of TelegraphCQ. Also on the roadmap is a distributed implementation. One form of distribution is the integration of TelegraphCQ with the TAG system [MFHH02] for aggregation over *ad hoc* sensor networks, but a further step that is planned is the distribution of the TelegraphCQ engine itself.

5 Related Work

There is a large volume of work related to the Telegraph project as a whole and to the underlying technology on which it is based. Such work is discussed in detail in the papers describing each of the individual Telegraph components. Here, we focus on projects that are related to the TelegraphCQ system.

Continuous queries were proposed by Terry et al. [TGNO92] for the purpose of filtering documents from a stream according to user requests specified in an SQL-like language. Seshadri et al.[SLR94] was another early effort to deal with the problem of defining and executing database-style queries over sequenced data.

NiagaraCQ [CDTW00] is an XML-based engine that supports continuous queries over changing data. NiagaraCQ builds static plans for the different continuous queries in the systems, and allows two queries to share a module if they have the same input. Bonnet et al.[BGS01, BS00] describe how devices can be modeled as ADTs in an extensible database, to allow different kinds of queries over them. They define three types of queries that can be posed over streaming data: historical, snapshot and long-running queries. STREAM [BW01, ABBM+02] is a data stream processing project whose focus is on computing approximate results and on understanding the memory requirements of posed queries. In particular, one of the project's goals is to understand how to efficiently run queries in a bounded amount of memory. The Aurora [CCCC+02] system allows users to specify quality-of-service requirements for queries, and then uses those specifications to determine how and when to shed load. TRIBECA [SH98] considers novel query modules over streams like multiplexers and demultiplexers.

Publish-subscribe systems are also related. SIFT [YF99] is a selective document dissemination system which allows users to subscribe to text documents by specifying a set of weighted keywords. It was one of the earlier projects to suggest the reversal of roles of queries and data in filtering systems through the use of an *inverted index* on the queries. Xfilter [AF00] and YFilter [DFFT02] are XML-document filtering engines that group and efficiently apply XPath queries over incoming documents. Fabret et al. [FJLP+01] observe that publish-subscribe systems can apply newly published events to existing subscriptions and can also match new subscriptions to existing events. Their solution, however, focuses only on the problem of grouping and optimizing subscription matching on the arrival of new data.

Other recent research has focussed on developing algorithms to perform specific functions on sequenced data. Lee et al. [LSM99] studies how to learn distributions from a stream and detect anomalies. Gehrke et al. [GKS01] considers the problem of computing correlated aggregate queries over streams, and presents techniques for obtaining approximate answers in a single pass. Yang et al. [YW01, YW00] discusses data structures for computing and maintaining aggregates over streams. Sadri et al.[SZZA01] propose SQL-TS, an extension of the SQL language to express sequence queries over time-series data.

Finally, our ideas on sharing work across queries are related to the problem of multi-query optimization. Originally posed by Sellis, et al., [Sell88] there has been a spate of work on this topic more recently, especially from the group at IIT-Bombay [RSSB00, DSRS01, GSV01]. Multi-query optimization typically shares relational sub-expressions that appear in the plans of multiple (snapshot) queries. In contrast, since TelegraphCQ shares modules

across multiple (continuous) queries, its ability to share work is more flexible; a similar point was made in by Madden et al., [MSHR02] in comparing CACQ to NiagaraCQ.

6 Conclusions and Future Work

The deployment of pervasive networking is leading towards a world where data is constantly in motion. In such a world, conventional techniques for query processing, which were developed under the assumption of a far more static and predictable computational environment, will not be sufficient. Instead, query processors, based on the idea of adaptive dataflow will be necessary. The Telegraph project has developed a suite of novel technologies for implementing continuously adaptive query processing.

In this paper we have described our ongoing implementation of the next generation Telegraph system, called TelegraphCQ. TelegraphCQ is focused on meeting the challenges that arise due to the need to handle large numbers of continuous queries over high-volume, highly-variable data streams. TelegraphCQ differs from other data stream and CQ projects due to its focus on extreme adaptivity and the novel infrastructure required to support such adaptivity. Our implementation is in an early stage and we have a long list of open research issues. However, our initial results are positive and based on our experiences with a succession of increasingly sophisticated prototypes, we believe that TelegraphCQ will be an excellent platform on which build applications and explore issues in deeply networked data management.

REFERENCES

[AF00] Altinel, M. and Franklin, M., Efficient Filtering of XML Documents for Selective Dissemination of Information. In VLDB (2000).

[AH00] Avnur, R., and Hellerstein, J., Eddies: Continuously Adaptive Query Processing. In SIGMOD (2000).

[Anto93] Antoshenkov, G., Dynamic Query Optimization in Rdb/VMS. In ICDE(1993).

[AAFZ95] Acharya, S., Alonso, R., Franklin, M., and Zdonik, S., Broadcast Disks: Data Management for Asymmetric Communications Environments. In SIGMOD (1995).

[BBDM+02] Babcock, B., et al., Models and Issues in Data Stream Systems. In PODS (2000).

[BGS01] Bonnet, P., Gehrke, J., and Seshadri, P., Towards Sensor Databases. In MDM (2001).

[BS00] Bonnet, P., and Seshadri, P., Device Database Systems. In ICDE (2000).

[BW01] Babu, S., and Widom, J. Continuous Queries Over Data Streams. *SIGMOD Record* (Sep 2001).

[CCCC+02] Carney, D., et al., Monitoring Streams - A New Class of Data Management Applications. In VLDB (2002).

[CDTW00] Chen, J., DeWitt, D., Tian, F., and Wang, Y., NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In SIGMOD (2000).

[CF02] Chandrasekaran, S., and Franklin, M., Streaming Queries over Streaming Data. In VLDB(2002).

[DG92] DeWitt, D., and Gray, J., Parallel Database Systems: The Future of High Performance Database Systems. *CACM* 35(6)(1992).

[DFFT02] Diao, Y., Fischer, P., Franklin, M., and To, R., YFilter: Efficient and Scalable Filtering of XML Documents (demonstration). In ICDE (2002).

[DNS91] DeWitt, D., Naughton, J., and Schneider, D. An Evaluation of Non-equijoin Algorithms. In VLDB (1991).

[DSRS01] Dalvi, N., Sanghai, S., Roy, P., Sudarshan, S., Pipelining in Multi-Query Optimization. In PODS 2001.

[FJLP+01] Fabret, F., et al., Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In SIGMOD (2002)

[GKS01] Gehrke, J., Korn, F., and Srivastava, D., On Computing Correlated Aggregates over Continual Data Streams. In SIGMOD (2001).

[Graf93] Graefe. G., Query Evaluation Techniques for Large Databases. *ACM Comp. Surveys* 25(2), June, 1993.

[GSV01] Gupta, A., Sudarshan, S., Viswanathan, S., Query Scheduling in Multi Query Optimization. In IDEAS 2001: 11-19.

[GW00] Goldman, R., and Widom, J., WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In SIGMOD (2000).

[HACO+99] Hellerstein, J., et al., Interactive Data Analysis with CONTROL, *IEEE Computer 32(8),* August, 1999.

[HFCD+00] Hellerstein, J., et al.,. Adaptive Query Processing: Technology in Evolution, *IEEE Data Engineering Bulletin,* June 2000.

[HHHL+02] Harren, M., et al., "Complex Queries in DHT-Based Peer-to-Peer Networks". In IPTPS (2002).

[HN96] Hellerstein, J., Naughton, J., Query Execution Techniques for Caching Expensive Methods. In SIGMOD(1996).

[KMCJ+00] Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M., The Click Modular Router. In *ACM TOCS* 18(3): (2000)

[MFHW02] Madden, S., Franklin, M., Hellerstien, J., Hong, W., A Tiny Aggregation Service for *ad hoc* Sensor Networks, In OSDI (2002).

[MF02] Madden, S., and Franklin, M., Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data, In ICDE (2002).

[MSHR02] Madden, S., Shah, M., Hellerstein, J., and Raman, V., Continuously Adaptive Continuous Queries over Streams. In SIGMOD (2002).

[RDH02] Raman, V., Deshpande, A., and Hellerstein, J., Using State Modules for Adaptive Query Processing, In ICDE (2003), *to appear*.

[RH02] Raman, V., Hellerstein, J., Partial Results for Online Query Processing. In SIGMOD(2002).

[RRH99] Raman, V., Raman, B., and Hellerstein, J., Online Dynamic Reordering for Interactive Data Processing. In VLDB (1999).

[RSSB00] Roy, P., Seshadri, A., Sudarshan, A., Bhobhe, S., Efficient and Extensible Algorithms For Multi Query Optimization. In SIGMOD(2000).

[Sell88] Sellis, T. "Multiple Query Optimization." *ACM TODS* 13(1):23-52, March 1988.

[SHCF03] Shah, M., Hellerstein, J., Chandrasekaran, S., and Franklin, M., Flux: An Adaptive Repartitioning Operator for Continuous Query Systems. In ICDE (2003), *to appear*.

[SMFH01] Shah, M., Madden, S., Franklin, M, and Hellerstein, J., Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System, *ACM SIGMOD Record,* 30(4), Dec, 2001.

[SH98] Sullivan, M., and Heybey, A., Tribeca: A System for Managing Large Databases of Network Traffic. In USENIX (1998).

[SLR94] Seshadri, P., Livny, M., and Ramakrishnan,R., Sequence Query Processing. In SIGMOD (1994).

[SZZA01] Sadri, R.., Zaniolo, C., Zarkesh, A., and Adibi, J., Optimization of Sequence Queries in Database Systems. In PODS (2001).

[TGNO92] Terry, D., Goldberg, D., Nichols, D,, and Oki, B., Continuous Queries Over Append-only Databases. In SIGMOD (1992).

[UF01] Urhan, T., and Franklin, M. XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Engineering Bulletin*, 23(2), 2000.

[UFA98] Urhan, T., Franklin, M., and Amsaleg, L., Cost Based Query Scrambling for Initial Delays. In SIGMOD (1998).

[WA91] Wilschut, A., and Apers, P., Dataflow Query Execution in a Parallel Main-memory Environment. In *Distributed and Parallel Databases* 1(1), 1993.

[YG99] Yan T. W., and Garcia-Molina, H. The SIFT Information Dissemination System. In *ACM TODS* 24(4), 1999.

[YW00] Yang, J., AND Widom, J., Temporal View Self-Maintenance. In EDBT (2000).

[YW01] Yang, J., AND Widom, J., Incremental Computation and Maintenance of Temporal Aggregates. In ICDE (2001).