Continuously Adaptive Continuous Queries over Streams

Samuel Madden
madden@cs.berkeley.edu

Mehul Shah
mashah@cs.berkeley.edu

Joseph M. Hellerstein
jmh@cs.berkeley.edu

Vijayshankar Raman
rshankar@cs.berkeley.edu

Abstract

We present a continuously adaptive, continuous query (CACQ) implementation based on the eddy query processing framework. We show that our design provides significant performance benefits over existing approaches to evaluating continuous queries, not only because of its adaptivity, but also because of the aggressive cross-query sharing of work and space that it enables. By breaking the abstraction of shared relational algebra expressions, our Telegraph CACQ implementation is able to share physical operators – both selections and join state – at a very fine grain. We augment these features with a grouped-filter index to simultaneously evaluate multiple selection predicates. We include measurements of the performance of our core system, along with a comparison to existing continuous query approaches.

1 Introduction

Traditional query processing operates in a request-response paradigm, in which a user poses a logical query against a database, and a query engine processes that query to generate a finite answer set. Recently, there has been interest in the *continuous query* paradigm, in which users register logical specifications of interest over streaming data sources, and a continuous query engine filters and synthesizes the data sources to deliver streaming, unbounded results to users (e.g., [22, 8]). An aspect of continuous query processing that has been overlooked in the literature to date is the need for adaptivity to change: unbounded queries will, by definition, run long enough to experience changes in system and data properties as well as system workload during their run. A continuous query engine should adapt gracefully to these changes, in order to ensure efficient processing over time.

With this motivation in mind, we used the Telegraph adaptive dataflow engine [15] as a platform for a continuous query engine; in this paper we discuss our continuous query implementation in Telegraph. We show how the eddy [4], a continuously adaptive query processing operator, can be applied to continuous queries. Our architecture, which we dub *Continuously Adaptive Continuous Queries (CACQ)*, offers significant performance and robustness gains relative to existing continuous query systems. Interestingly, our scheme provides benefits even in scenarios where no change is evident, due to its ability to share computation and storage across queries much more aggressively than earlier approaches that used static query plans.

Our interest in continuous queries arose in the context of our work on handling streams of data from sensor networks [23]. Researchers from the TinyOS and SmartDust projects at UC Berkeley or the Oxygen project at MIT [16, 19, 14] predict that our environments will soon be teeming with tens of thousands of small, low-power, wireless sensors. Each of these devices will produce a stream of data, and those streams will need to be monitored and combined to detect interesting changes in the environment.

To clarify the techniques presented in this paper, we consider a scenario from sensor networks. One application of sensor networks is *building monitoring*, where a variety of sensors such as light, temperature, sound, vibration, structural strain, and magnetic field are distributed throughout a building to allow occupants and supervisors of that building to monitor environmental properties or human activity. For instance, structural engineers might wish to use vibration sensors to detect earthquakes and strain sensors to assess structural integrity. Employees might wish to use light or motion sensors to tell if their boss is in her office. Building managers could use temperature and motion readings to automatically adjust heating and lighting. Autonomous devices such as lighting systems, door locks, sprinkler systems and window shades could register queries to drive their behavior. We assume that for each distinct type of sensor there is one logical sensor-reading "table" or data source that accumulates the readings from all the sensors of that type. Each entry in a readings table contains a sensor id, a timestamp, and a sensor value. In a large office building, there might be several thousand such sensors feeding dozens of logical tables, with thousands of continuous queries.

This scenario illustrates the requirements of our continuous query system: there are numerous long running queries posed over a number of unbounded streams of sensor readings. As sensor readings arrive, queries currently in the system must be applied to them, and updates to queries must be disseminated to the users who registered the queries. Users can pose or cancel queries at any time, so the operations that must be applied to any given tuple vary depending on the current set of queries in the system.

Our CACQ design incorporates four significant innovations that make it better suited to continuous query processing over streams than other continuous query systems. First, we use the eddy operator to provide continuous adaptivity to the changing query workload, data delivery rates, and overall system performance. Second, we explicitly encode the work which has been performed on a tuple, its *lineage*, within the tuple, allowing operators from many queries to be applied to a single tuple. Third, we use an efficient *predicate index* for applying many different selections to a single tuple. Finally, we split joins into unary operators called *SteMs* (State Modules) that allow pipelined join computation and sharing of state between joins in different queries. The next section motivates each of these techniques with specific examples. Following this, we present details of our implementation in Section 3. Section 4 discusses routing policies for CACQ. Section 5 compares our continuous query system to other continuous query implementations. Finally, Section 6 discusses related work and Section 7 presents future work and conclusions.

2 Challenges and Contributions

The challenge in designing a continuous query system is to minimize the amount of storage and computation that is required to satisfy many simultaneous queries running in the system. Given thousands of queries over dozens of logical sources, queries will overlap significantly in the data sources they require. It is highly likely that queries over the same source will contain selection predicates over overlapping ranges of attributes, or request that the same pairs of sources be joined. To efficiently process the outstanding queries, the continuous query processor must leverage this overlap as much as possible. Query processing is further complicated by the long running nature of continuous queries: query cost estimates that were sound when a query was first posed may be dead wrong by the time the query is removed from the system. In this section we discuss the four main contributions of CACQ for addressing these challenges.

2.1 Adapting to Long Running Queries

To illustrate the problems that can arise when a static query optimizer is used to build query plans for long running queries, consider an example from our building monitoring scenario: many queries may request building locations where the lights are on, since illuminated rooms correspond to areas that are occupied by people. Thus, many queries will include a selection predicate looking for light levels above some threshold. During normal working hours, this predicate is not very selective: most areas of the building are lit. Thus, a static query optimizer would normally place the predicate towards the top of the query plan. However, at night, few locations are lit (or occupied), so this becomes a very selective predicate that should be pushed to the bottom of the plan. A static optimizer cannot easily change its decision; it is possible that the optimizer could be periodically re-run, but deciding when to do so would be complicated. Moreover, it is difficult in a traditional query engine – including one designed for continuous queries – to modify the order of operations in a query plan while the query is in flight. Eddies circumvent this problem through continuous adaptivity: the route that a tuple takes through operators in a query is dynamically chosen so that tuples which arrive during working hours can have operators applied in a different order than tuples that arrive at night. In order to enable this flexible routing, a system that uses eddies by necessity incorporates query processing algorithms that are amenable to in-flight reordering of operations [4]. The eddy determines the order in which to apply operators by observing their recent cost and selectivity and routing tuples accordingly. The basic mechanism for implementing continuous adaptivity is discussed in Section 3.1 below.

2.2 Explicit Tuple Lineage

As a result of the reordering endemic to eddies, the path that each tuple takes through the operators – its *lineage* – is explicitly encoded in the tuple. Different tuples accumulate different lineages over time, but in the end an eddy produces a correct query result. Note that a query processing operator connected to an eddy – for example, a pipelined hash join [40] – may process tuples with different lineages, depending on whether or not they have been previously routed through selections, other joins, etc. This contrasts with systems based on static query plans, in which the state of intermediate tuples is implicit in the query plan. Query operators in a static plan operate on tuples of a single lineage. Because Telegraph is designed to work correctly with eddies, its query operators correctly handle tuples within a query that have multiple different lineages. In CACQ we extend this ability to multiple overlapping queries, maximizing the sharing of work and state across queries.

As an example, consider a number of queries over our building network, each of which is looking to temporally join temperature and light sensor readings above some threshold, with the light threshold varying from query to query. Each query consists of two operators: a selection over light readings and a windowed join [33] within some time window between the two sets of readings. All queries have the same join predicate, but each query selects a different set of light tuples (some of which satisfy multiple queries). Our CQ goals dictate that we should try to share work whenever possible; since all queries contain a join with an identical predicate (equality join on time between light and temperature tuples), an obvious trick would be to perform only a single join.

By tracking a tuple's lineage, we are able share the work of the join operator between all queries, even when the selections for each query are applied before the join. Lineage is encoded in a bit vector kept with each light tuple as it flows through the selections from the various queries and the single unified join operator. Each query's selection operator sets a single bit in the tuple corresponding to whether the selection accepts or rejects the tuple. If a tuple is rejected by the selections from all queries, the eddy can discard the tuple. Otherwise, the tuple can be routed into the join, which locates matching temperature tuples. Tuples output from the join are then sent to the queries whose query bits indicate the tuple was accepted by its selection operator. Thus, only a single join operator is needed, but selection predicates can be applied first. A detailed discussion of our techniques for maintaining a tuple's lineage is provided in Section 3.2 below. Joins are discussed in Section 3.3.

2.3 Grouped Filter: Selection Predicate Indexing

Our third technique for continuous query processing is a predicate indexing operator called a *grouped filter* that reduces computation when selection predicates have commonalities. We maintain a grouped-filter index for each attribute of each source that appears in a query, and use that index to efficiently compute overlapping portions of range queries. Consider our building monitoring scenario again: different users may have different preferences for temperature in their offices, and the central heating system may use the sensor network to determine temperature in those offices. The heating system could decide to direct heat to a particular part of the building by posing a number of continuous queries looking for temperatures under the user-specified threshold in each office. Each query is thus a pair of selections on location and temperature. It is very likely that temperature predicates will overlap: the comfort range for most people is fairly similar. Thus, with an index over temperature predicates, we can avoid applying each predicate independently: we ask the grouped-filter to find all predicates requesting a temperature above the value of the tuple's temperature field. The tuple is then filtered by building location and output to the queries that match, which the heating system uses to adjust the building temperature in the appropriate location. Details of the grouped filter are discussed in Section 3.2.6; for now, it should be thought of as an opaque object that takes in multiple predicates and a tuple, and efficiently returns the set of predicates that accept the tuple.

2.4 SteMs: Multiway Pipelined Joins

Users may issue queries that join data from distinct but overlapping subsets of the sources. For example, continuing with our building monitoring scenario, imagine that one user wants the blinds in a region of the the building to close if it is both warm and sunny at the same time, while another user wants the windows to open if it is both warm and quiet at the same time. Assume that readings from the temperature, light, and sound sensors are all tagged with a `location` and `time` and arrive in time order. As new data arrives, a continuous query system must compute a join on the `location` and `time` attributes over these sources and stream results to clients so they can react quickly to changing conditions. Moreover, a continuous query system must simultaneously handle numerous such queries that have varying overlap in the set of sources they combine.

To fulfill these requirements for computing joins over streaming (and non-streaming) sources, our CACQ system employs two techniques. First, we modify our notion of join in a standard way [33]: tuples can only join if they co-occur in a time *window*. This modification bounds the state we need to maintain to compute joins. Second, we use a space-efficient generalization of doubly-pipelined joins [40] within our eddy framework. For each incoming source, we build an index on-the-fly and encapsulate the index in a unary operator called a SteM, introduced in [28]. SteMs are exposed to the eddy as first class operators, and the eddy encapsulates the logic for computing joins over the incoming sources using SteMs. This technique permits us to perform a multiway pipelined join. That is, it allows us to incrementally compute a join over any subset of the sources and stream the results to the user. Moreover, this technique allows us to share the state used for computing joins across numerous queries. We describe the details of this scheme in Section 3.

In the following section, we discuss the implementation of our CACQ system in detail, focusing on the four techniques presented above. After a brief introduction to our query engine, Section 3.1 begins with a discussion of processing a single query over a single data source. Section 3.2 shows the new data structures and tuple format required to extend this system for multiple queries each ranging over only a single data source. In Section 3.2.6 we discuss how predicate indices can be used to save work in overlapping predicates. Finally, in Section 3.3, we show how to extend the system with join queries over multiple sources, and discuss the implementation of SteMs.

3 Implementation

We implemented our CACQ system in the context of the Telegraph query processing engine that has been developed over the past two years by the UC Berkeley Database group [15]. It supports read-only SQL-style queries (without nested queries) over a variety of data sources: files, network and sensor streams, and web pages. Streams are treated as infinite relational data sources, and web pages are mapped into relational tables via simple, regular-expression based-wrappers [21]. Instead of a conventional query plan, Telegraph uses the eddy operator to dynamically routes tuples arriving from data sources into operators that operate on those tuples. Telegraph provides operators to perform basic dataflow operations like select and join.

Given Telegraph as our development platform, we now discuss our CACQ implementation. We will describe techniques to fully implement select-project-join (SPJ) queries without nesting or aggregation. We provide a brief discussion of aggregation in Section 7. In this work, we only describe queries over streaming data. It is assumed that queries apply to data present in the system from the moment the query is registered and any future data which may appear until the query is removed from the system. Queries over historical or non-streaming data are not a part of this implementation, although we will turn to them briefly in the Section 6 on related work.

Throughout this work, we map stream elements such as sensor readings onto relations, as proposed in [27]. This allows queries posed over streaming data to refer to relations and relational attributes. This mapping is done in the obvious way: each field of a stream-element corresponds to an attribute of the relation representing that stream. We assume that each stream element has the same fields, as well as a time stamp that indicates when it was produced.

Given these caveats, we now present the design of our system. For clarity of exposition, we consider designs in increasing order of complexity. We begin with a rudimentary CACQ system in which a single query without joins runs over a single source with multiple attributes. We then show how multiple queries without joins can be processed simultaneously, sharing tuples and selection operators. Finally, we show how joins can be added to the system and, how they share state via SteMs.

3.1 Single Query without Joins

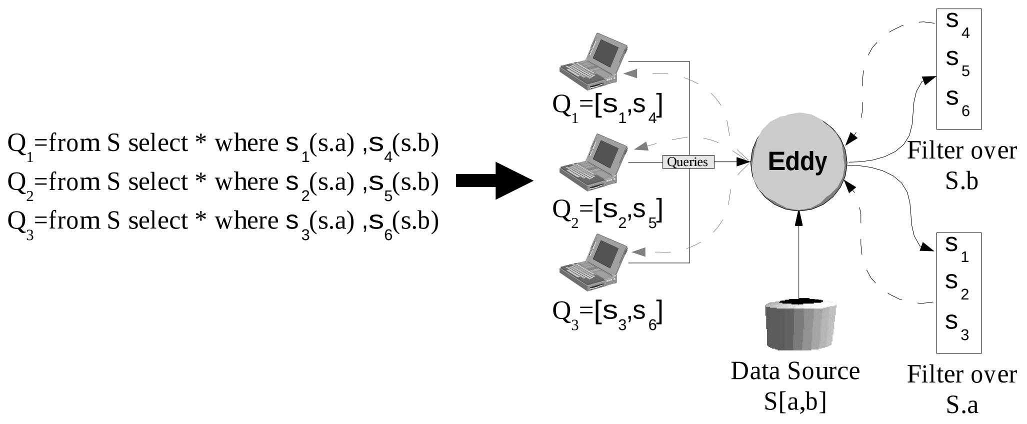
With only a single query, the CACQ system is very similar to Telegraph with a standard eddy operator, as in [4]. The query is decomposed into a set of operators that constitute the processing that must be applied to every tuple flowing into the system. Since we are not considering joins at the moment, the only operators that can exist are *scan operators*, which fetch tuples, and *selection operators*, which filter those tuples based on a user-specified Boolean predicate. For now, we assume that queries contain only conjunctions (ANDs) of predicates; we discuss disjunctive predicates (ORs) in Section 3.2.7.

At the core of the system is a single eddy that routes tuples to operators for processing. Each operator has an input queue of tuples waiting to be processed by it. Operators dequeue tuples, process them, and return them to the eddy for further routing. The eddy maintains a pool of tuples that are waiting to be placed on the input queue of some operator. When the pool is empty, the eddy can schedule a scan operator to cause more tuples to be fetched or produced. Notice that the eddy can vary the route a tuple takes through operators in the system on a per-tuple basis. Also note that tuples are never copied: once allocated, they are passed by reference between operators. For a more detailed discussion of Telegraph, see [34].

3.1.1 Routing in the Single Query Case

As in [4], to facilitate routing, the eddy maintains two bit vectors with each tuple. Each bit vector contains a number of bits equal to the number of operators in the system. These vectors are used to track the operators which have or may still be applied to a tuple. The first, the `ready` bits, indicate which operators can be applied to a tuple. In the single table case, any tuple can be routed to any operator, so the `ready` bits are initially all set. The second bit vector is contains the `done` bits that indicate the operators to which a tuple has already been routed. Initially all of the `done` bits are cleared. Once all of a tuple's `done` bits have been set, it can be output. For the simple selection case, the done bits are the complement of the ready bits: once a tuple has been processed by a particular operator, that operator's `ready` bit is cleared and its `done` bit is set. In Section 3.3, we will see cases where the two bitmaps will not be complements of each other.

Our only query processing operator so far, the selection operator, uses these bits as follows: when a tuple arrives, it applies its predicate to the tuple. If the tuple does not satisfy the selection predicate, the operator discards the tuple by deallocating it and not returning it to the eddy. If the tuple satisfies the predicate, the operator's `done` bit is set and



Q1=from S select * where s1(s.a) ,s4(s.b)
Q2=from S select * where s2(s.a) ,s5(s.b)
Q3=from S select * where s3(s.a) ,s6(s.b)
Q1=[s1,s4]
Q2=[s2,s5]
Q3=[s3,s6]
Queries
Eddy
s4
s5
s6
Filter over S.b
s1
s2
s3
Filter over S.a
Data Source S[a,b]

Figure 1: *The Basic Continuous Query Architecture*

its `ready` bit is cleared; the tuple is then returned to the eddy for further processing. The total storage overhead of these vectors, in bits per tuple, is twice the number of operators in the query.

The final element of this simple single query CACQ system is a way to determine the order in which tuples are routed through operators. This is a policy decision: any ordering will eventually result in a tuple being fully processed, but some orderings, such as those which place highly selective selections earlier in routing, will be more efficient than others. The eddy employs a *routing policy* to choose the tuple to route and the next operator to route it to. The routing policy implements the per-tuple adaptivity of eddies. In the query case, assuming all selections cost the same to apply, the policy should route to more selective operators first. We discuss routing polices in our CACQ system in Section 4.

Given this simple single query CACQ approach, we now describe how to extend this solution to work with multiple queries, each of which queries a single source.

3.2 Multiple Queries without Joins

The goal of the multiple-query solution in the absence of joins is to use a single eddy to route tuples among all of the continuous queries currently in the system. In our solution, tuples are never copied: two different queries with two different predicates over the same relation should operate over exactly the same tuples. This is important for two reasons: tuples occupy storage that must be conserved, and copying takes valuable processor cycles. A key part of the multiple query solution without joins is the grouped filter that allows us to share work between multiple selections over the attribute of a relation. We present the design of this data structure in Section 3.2.6 below.

Figure 1 shows the basic architecture. Users submit queries Q_1, Q_2, and Q_3, consisting of selection predicates S_1 through S_6 over two fields, a and b of source S. All queries are submitted to a single eddy, with just one filter operator (and its associated grouped filter) for field a, and one for field b. The eddy tracks when tuples are ready to be output to each query, and sends the tuples back to the appropriate end-users as required. We will refer to this single eddy, and the operators associated with all queries running into it as a *flow*. The rest of this Section uses this example to show how queries are added and tuples routed through flows without joins.

There are two modifications that must be made to the single query system to allow it to handle multiple queries: new queries must be combined with old queries, sharing operators wherever possible, and tuples must be properly routed through the merged set of queries and output to the appropriate end users.

3.2.1 Adding a Query

Combining old and new queries is not complicated. A new query which scans the same relation as an existing query will share existing scan operator. Similarly, a new query with a selection over some attribute a for which a grouped filter already exists will simply add its predicate over a to the filter.

As an example, consider the case where a second query is added to an eddy that already has a single query over a single source. We'll begin with Q_1 from Figure 1. By itself, this query consists of three operators: a scan on S and two selections, one over $S.a$ and one over $S.b$. Now, Q_2 arrives, which also contains a scan on S, a selection over $S.a$ and a selection over $S.b$. We begin by matching the scans: both are over the same relation, so we do not need to instantiate a new scan for Q_2. Similarly, we can add the predicates from the selections in Q_2 to the grouped filters created over $S.a$ and $S.b$ when instantiating Q_1. (Remember, we are not considering the case where Q_2 is interested in historical tuples; if this were the case, we would have to create a different scan operator.)

3.2.2 Routing in the Multiple Query Case

We now turn to routing tuples through a flow. We use the same approach as in the single query case: the eddy repeatedly uses its routing policy to choose a tuple to route and an to which operator the tuple should be routed. A complexity arises when routing through a predicate index. When S_1 accepts a tuple and S_2 rejects it, we need to record this information somewhere, since it means that the tuple must not be output to Q_2, but might be output to Q_1 (if S_4 does not reject it.)

Our solution is to encode information about queries that accept or reject a tuple in the tuple itself, just as we already store the `ready` and `done` bits with the tuple. We allocate a bitmap, `queriesCompleted`, with one bit per query, and store it in the tuple. When a query's bit is set, it indicates that this tuple has already been output or rejected by the query, so the tuple does not need to be output to that query. Thus, Q_2 will have its bit turned on in this bitmask when S_2 rejects a tuple, and Q_1 will have its bit turned on when a tuple is output to it.

The `queriesCompleted` bitmap, along with the `ready` and `done` bits, completely encode the notion of a tuple's lineage discussed above. Lineage does not simply capture a tuple's path through a single query: it concisely expresses a tuple's path through all queries in the system. By looking at any tuple at any point in the flow, it is possible to determine where that tuple has been (via its `done` bits), where it must go next (via its `ready` bits) and, most importantly, where it may still be output (via its `queriesCompleted` bits.) In our CACQ approach, we are never dependent on the structure of the query plan for implicit information about a tuple's *lineage*. This means that any operator common to any two queries can be treated as a single operator that handles tuples for both queries. Similarly, any tuple common to two queries can be used without copying. The fact that the tuple may be output to only one of the queries is explicitly encoded in the `queriesCompleted` bits. In existing continuous query systems (like NiagaraCQ [8]) that use a static query plan, a pair of operators that could otherwise be merged must be kept separate if the query optimizer cannot guarantee that the set of tuples flowing into one operator is identical to the set of tuples flowing into the other. Again, this leads to extra copies of each tuple which would not be allocated in the CACQ approach.

3.2.3 Outputting Tuples

We have now shown how to track the queries to which a tuple may or may not need to be output to, but a mechanism to determine *when* a tuple should be output is still needed. We will this by associating a compact query signature, the `completionMask` , with each query. The completion bitmask is the same size as the `done` bitmap, and has a bit turned on for each operator that must be applied to the tuple before it can be output. To determine if a tuple t should be output to a query q that has not already output or rejected t, we AND q's `completionMask` with t's `done` bits; if the value is equal to the `completionMask`, the tuple can be output to q. We maintain a separate data structure,

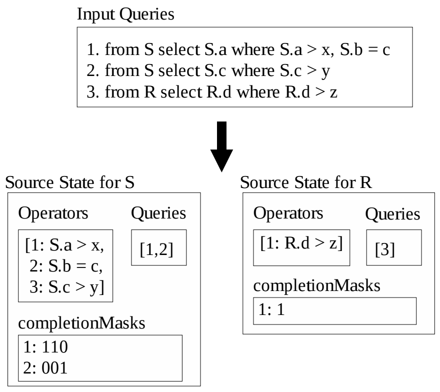

Input Queries
1. from S select S.a where S.a > x, S.b = c
2. from S select S.c where S.c > y
3. from R select R.d where R.d > z
Source State for S
Operators
Queries
[1: S.a > x, 2: S.b = c, 3: S.c > y]
[1,2]
completionMasks
1: 110
2: 001
Source State for R
Operators
Queries
[1: R.d > z]
[3]
completionMasks
1: 1

Figure 2: *Continuous Query Data Structures*

`outQueues`, which a query ID to an output queue that will deliver tuples to the user who posed each query.

The above system will properly merge together queries and route tuples through them. There is however, an optimization that significantly improves the space efficiency of this approach. Consider what happens when a new query, Q_4, with a single selection over some source R is added to the queries shown in Figure 1. This query shares no operators with the other queries in the system and an R tuple will never be routed through one of the selection operators on S, but space must be reserved in every R tuple for the `done` and `ready` bits of the selections on S and in every R tuple's `queriesCompleted` bits for Q_1,Q_2, and Q_3. In a system with many queries over many sources, this could lead to a significant waste of space in every tuple.

The solution is to partition our state by source. State that was previously system-wide, namely information about queries and operators, now becomes specific to a particular data source. Each tuple is tagged with a `sourceId` which tells the eddy which scan operator created a tuple and is used to determine how that tuple's `done`, `ready`, and `queriesCompleted` bits are interpreted. Two auxiliary data structures are created for each `sourceId`: an `operators` table that lists the operators that apply to the source and a `queries` table that lists the queries over the source. The ith entry in the `operators` list corresponds to the ith bit in the `done` and `ready` bitmasks for tuples with this `sourceId.` Similarly, entries in `queries` correspond to bits in a tuple's `queriesCompleted` bitmap. We must also insure that the `completionMasks` built above are associated with the appropriate source's `operators` list. Figure 2 shows these data structures for three sample queries.

Figure 3 shows the extra fields stored in the tuple header for routing in CACQ eddies. The fields that are inherited from the single-query tuple are shown first, with the additional fields for the multi-query case shown next.

Even with our source-tagging optimization, these data structures impose a storage overhead on our CACQ system.

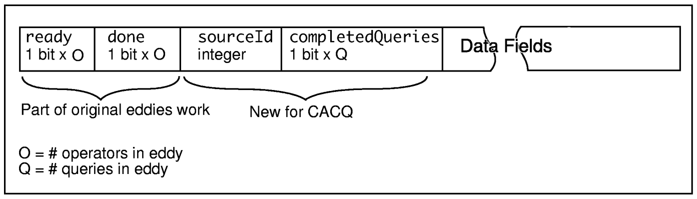


Figure 3: *Continuous Query Tuple Format*

Table 1: **Continuous Query Storage.** *Extra data structures required for continuous queries a Flow, with estimates for 100,000 Queries*

	Structure	Size Expression	Estimated Size (bytes)
Per Source State (SS)	`queries`	sizeof$(int) \times \|S_{queries}\|$	80,000
	`operators`	sizeof$(Operator) \times (\|S_{selections}\| + 1(scan))$	400
	`completionMask`	$\|S_{queries}\| \times \frac{\|S_{selections}\|}{8}$	7,500
Per Tuple State (TS)	`tupleQueryMask`	$\frac{\|S_{queries}\|}{8}$	2,500
Per Query State (QS)	`outQueues`	sizeof$(OutputQueue)$	64
Total Flow State	$\|sources\| \times$ SS + $\|queries\| \times$ QS + $\|activetuples\| \times$ TS		6.83 MB + $\|activetuples\| \times$ 2.5kB

In the next Section we estimate the magnitude of this overhead and present encouraging experimental results showing that the performance penalty incurred by maintaining the additional per-source and per-tuple state is reasonable.

3.2.4 Performance Effects of Additional Storage

We begin this Section by estimating the amount of storage required to maintain the per-tuple, per-source, and per-query state in our CACQ system.

Table 1 summarizes the storage overhead of these additional data structures (not including the `ready` and `done` bits from the basic Eddies implementation.) In this table, $|S_{selections}|$ refers to the number of distinct selection operators reachable per source; this is the total number of predicate indices (equal to the number of attributes), divided by the total number of queried sources. $|S_{queries}|$ refers to the average number of queries that involve tuples from this source; in the absence of joins this is equal to the average number of queries divided by the number of sources. Table 2 gives representative values from our building monitoring scenario for these parameters, assuming the (very aggressive) goal of 100,000 simultaneous queries. These are used to provide numeric values in Table 1. Notice that the additional cost of continuous queries is just 6.83MB, the majority of which is the output queues for queries. The state per active tuple is 2.5kB, which seems troublesome until one considers the case in which 100,000 queries are run independently. In this case, 100,000 copies of each tuple will be made, which for 200 byte tuples is 20MB of state required for all copies of each tuple.

In the next Section, we present experimental evidence showing how query performance relates to tuple size in the Telegraph query engine.

Table 2: **Continuous Query Estimated Parameters for Building Monitoring Scenario.**

Parameter	Value
$\|queries\|$	100,000
$\|sources\|$	5 (light, temperature, sound, vibration, magnetic)
$\|attributes\|$	15 (3 attributes per source)
$\|predicates\|$	150,000 (avg. 1.5 filters / query)
$\|S_{selections}\|$	3
$\|S_{queries}\|$	20,000
sizeof(int)	4 bytes
sizeof$(OutputQueue)$	64 bytes
sizeof$(Predicate)$	100 bytes
sizeof$(Operator)$	100 bytes

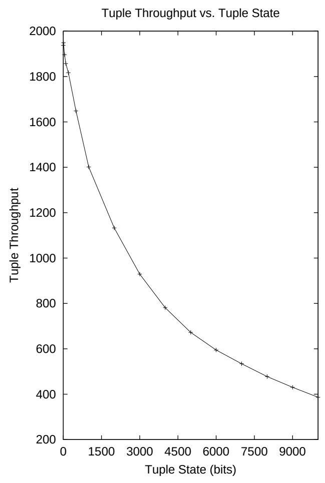


Figure 4: *Eddy Performance Vs. Tuple Size*

3.2.5 Storage Overhead

In order for the system to be able to scale to an arbitrary number of queries, we must be able to extend the size of tuples without seriously impacting performance. It is expected that larger tuples will lead to slower performance, simply because more bits have to be allocated and copied every time a tuple is operated upon.

All of our experimental results are generated from the actual Telegraph query engine running in real time. As a continuous data source, we use a generated stream S of random tuples, each with six fields: an sequence number uniquely identifying the tuple and five integer valued fields, S.a through S.e with random values uniformly sampled from the range [0,100).

The server was an unloaded Pentium III 933 MHz with 256 megabytes of RAM. Telegraph was running under the Sun HotSpot JDK 1.3.1, on Debian Linux with a 2.4.2 Kernel. Client connections came from a separate machine, running HotSpot JDK 1.3. To avoid variations in network latency, tuples were counted and discarded just before being sent back to the client machine.

In these studies, we ran 5 simultaneous queries over the source S described above. We varied the tuple state size from 15 bits/tuple (the minimum required for 5 queries) to 3000 bits/tuple (the default value used for other experiments) and measured the tuple throughput rate. We purposely kept the number of queries small to measure the impact of additional tuple state independently from the cost of additional queries and operators, whose performance we will discuss in Section 3.2.8. The results are shown in Figure 4. Notice that tuple throughput drops off by more than a factor of three between 15 and 3000 bits per tuple, but that the slope of the curve is decreasing, such that adding many more bits will not significantly decrease query performance. In fact, the tail of the graph is proportional to $\frac{1}{\#bits}$. This represents the memory bandwidth of our system: there is a fixed number of bytes per second we can allocate and format for tuples. Longer tuples require more bytes, and so fewer of them can be allocated and formatted in one second.

These results demonstrate although tuple size does dramatically effect performance, it does not cripple our query processor. It is important to bear in mind that this amount of source state is enough to run one thousand simultaneous queries – an amount of parallelism that would severely stress any other database system.

Having summarized the costs of our extra tuple state, we now present in detail our predicate index for efficiently grouping selection predicates.

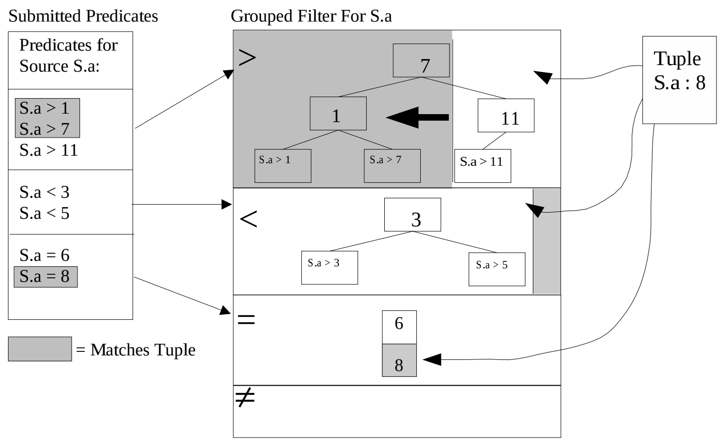


Figure 5: *Grouped Filter Example:* The grouped filter is searched for matching predicates when a tuple arrives. The grayed regions correspond to matching predicates for the tuple in the upper right.

3.2.6 Predicate Index: The Grouped Filter

As previously mentioned, our CACQ system includes a predicate index that allows us to group selection predicates, combining all selections over a single field into a *grouped-filter* operator, which can apply many predicates to a single tuple more efficiently than applying each predicate independently.

When a selection operator is encountered in a new query, its source field is checked to see if it matches an already instantiated grouped filter. If so, its predicate is merged into that filter. Otherwise, a new grouped filter is created with just a single predicate.

A grouped filter consists of four data structures: a greater-than balanced binary tree, a less-than tree, an equality hash-table, and an inequality hash-table. When a new predicate arrives, it is inserted into the appropriate data structure (i.e. $>$ predicates are put into the greater-than tree) at the location specified by its constant value; greater-than-or-equal-to and less-than-or-equal-to predicates are inserted into both a tree and the equality hash-table. Note that we do not have to store the entire predicate data structure: we keep the predicate's constant value and the query-id of the query it applies to (in particular, we do not need to store the database table or field the predicate applies to.)

When a tuple arrives at the filter, each of these data structures is probed with the value of the tuple. For the greater-than tree, all predicates that are to the left of the value of the tuple are matches; likewise, for the less-than-tree, all predicates to the right of the value of the tuple are matches (see Figure 5.) For the equality hash, a match only occurs if the value of the tuple is in the table. Conversely, in the inequality case, all tuples are matches *except* those that appear in the table.

As matches are retrieved, a bit-mask of queries is marked with those queries whose predicates the tuple passes. Once all of the matches have been found, the mask is scanned, and the tuple's `queriesCompleted` bitmap is modified to indicate that the tuple should not be output to those queries which the tuple did *not* pass – in this way, these queries are prevented from seeing non-matching tuples.

Figure 5 illustrates these data structures for a set of predicates over a single field. A probe tuple is shown on the right, and the gray boxes indicate the matching predicates within the data structures.

In addition to significantly reducing the number of predicates that must be evaluated when many predicates exist over a single field, grouped predicates are interesting for another reason: they represent a significant reduction in the number of operators through which an eddy must route a typical tuple. This provides a number of benefits: First, it serves to reduce the average tuple size, since tuples need fewer operator-bits in their headers. Second, it reduces the size of the operator-state stored with each source. Finally, it eliminates a large number of routing steps in the flow of the tuple: even though each step is not particularly expensive, routing a tuple through thousands of filters will incur a non-trivial routing overhead.

We now briefly consider how queries with disjunction (ORs) of selection predicates work.

3.2.7 Queries with Disjunction

Up to this point, we have only considered queries with AND predicates. To handle ORs, we follow standard practice [31] and reduce boolean expression into conjunctive normal form, for example:

$(S_1 \vee S_2) \wedge (S_3) \wedge (S_4 \vee S_5)$

The eddy is still free to choose the order in which tuples are routed through the selection operators or grouped filters. Also, eddy applies the same check to determine if a tuple needs to be output to a query. But, we modify our procedure in the operators for adjusting the `ready` and `done` bits and for determining when a tuple is rejected by a query. If any of the terms in a conjunct evaluates to true then the entire conjunct is true, so there is no need to evaluate the other terms. In this case, the `done` bits corresponding to every predicate in the conjunct are all set, and the corresponding `ready` bits are cleared. However, if a term evaluates to false, we need to apply the other predicates in the conjunct, so only the `done` and `ready` bits corresponding to that predicate are modified. If the last term in a conjunct evaluates to false, then we set all the `done` bits and clear the `ready` bits for all the predicates and mark the query as completed. We omit a detailed description of how the above logic is evaluated due to lack of space.

Thus, our CACQ system fully supports expressions containing ANDs and ORs of selection predicates. We now turn to a performance assessment of this complete multi-query, single source system.

3.2.8 Performance of CACQ without Joins

One of the stated goals for our system was to allow it to scale to a large number of simultaneous queries over a number of data sources. We believe our system has made significant progress towards that goal. To demonstrate this, we ran two experiments: In the first, we measured the effect of increasing the number of queries; In the second, we varied the number of data sources over which those queries were posed.

Queries in both scenarios were randomly generated. Randomly generated queries had a 50% chance of having a predicate over any field; if a predicate existed for a given field, the predicate was randomly selected from the set $<, >, \leq, \geq$. Equality and inequality predicates were omitted because randomly generated equality queries will rarely overlap. The comparison value was a random uniform selection from the range [0, 100).

To measure the performance of our CACQ implementation against the number of queries, we issued many queries against a single data source and measured the number of tuples output from the system. We compared the performance of continuous queries against the basic implementation in Telegraph, in which each query runs with its own eddy and operators. Figure 6 shows the results from these experiments. Notice that for the continuous query case, throughput increases sharply to about 20 queries, at which point the system is fully utilized; the system cannot handle more queries without decreasing the per-query delivery rate. It continues to scale at this throughput rate to fifty queries and beyond. The existing eddy implementation reaches maximum throughput at five queries, with a total tuple throughput of about half of our continuous query system.

To measure the ability of our system to scale to a large number of sources, we experimented with running twenty queries over a variable number of sources identical to the S source described above. Each query had a single predicate randomly selected as above, all over field a, with the source for each query randomly chosen from the available sources. Multiple queries were issued over the same source. Figure 7 plots the number of tuples output versus the number of sources. As expected, additional sources decrease the tuple throughput somewhat. This is due to two factors: first, there are now many more scanner operators that must be scheduled by the eddy. Second, because filters of independent

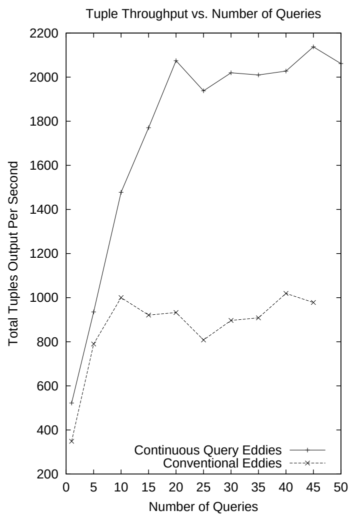


Figure 6: *Eddy Performance Vs. Number of Queries*

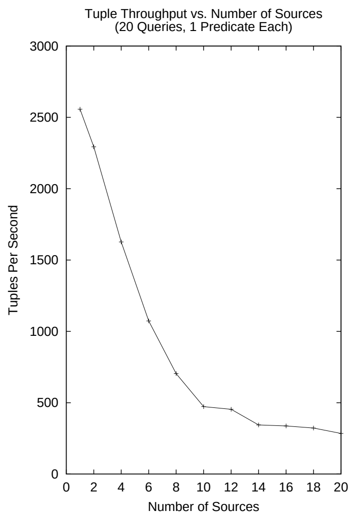


Figure 7: *Eddy Performance Vs. Number of Sources*

streams cannot be combined, many more filter-operators are created and a larger number of predicates evaluated as more sources are added. Notice, however, that the system scales reasonably with the number of sources: by twenty sources, performance has only dropped by about a factor of two.

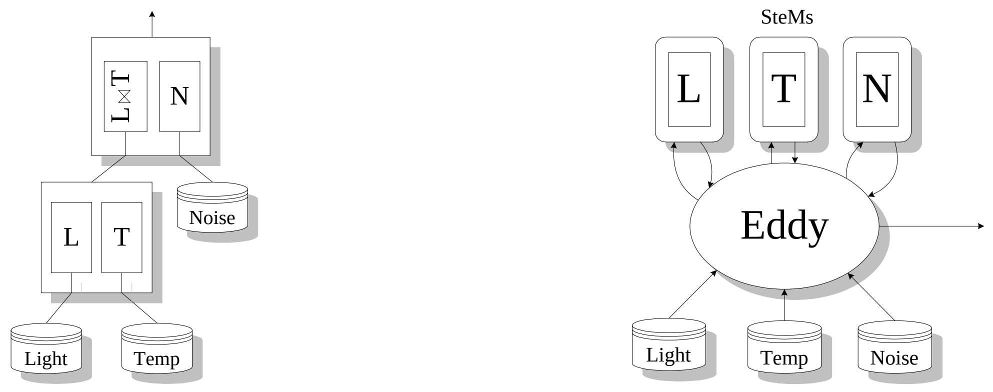
3.3 Multiple Queries with Joins

Thus far, we have only presented queries containing selection operators. In this section, we present our mechanism for computing joins over streaming sources. As mentioned before, we have two requirements for join processing in our CACQ system. First, we must insure that the join operations are pipelined to support continuous streaming results so that users may receive updates quickly. Second, we must scale with the number of queries, where each query can specify a join and predicates over any subset of the sources. To accomplish these goals, we use a generalization, within our eddy framework, of doubly-pipelined hash-joins called SteMs [28] which allows multiway-pipelined join computation for any subset of the incoming sources. This scheme reduces the state needed for join computation by sharing the in-flight index structures built among the various joins specified. Furthermore, we show how to extend this scheme to share intermediate join results among the queries specified. Finally, since the in-flight indices are built over possibly infinite streams, we describe how we purge these structures of unneeded, stale state for the join computation to bound the space consumed.

3.3.1 SteMs: Multiway-Pipelined Joins

One goal of our CACQ system is to allow users to quickly react to changing conditions in their input data. Thus, we ensure our computation is pipelined, so we can quickly produce new results from the data collected so far when a new tuple arrives from any source. These results must be produced incrementally for all queries, not just some of the specified queries. We fulfill these requirements by using a space-efficient generalization of doubly-pipelined hash joins within our eddy framework called SteMs. SteMs were first developed in [28] in the context of adaptive query processing.

First, let us review doubly-pipelined joins, their properties, and why cascades of such joins can be inefficient. A doubly-pipelined hash join is a binary join which maintains an in-flight hash index on each of its input relations, call



(a) A Tree of Doubly-Pipelined Hash Joins (b) Eddy and SteMs

Figure 8: Conventional Query Plans vs. CACQ

them R and S. When a new tuple arrives from one of the input relations, say R, it is first inserted into the index for R, and then used to probe the index of S for matches. Note, both the insertion and probe phase for one tuple completes before the next tuple can be processed. In order to build pipelined joins over more than two sources, we can compose such joins into a tree of joins, exactly as one would in a static query plan. An example is shown in Figure 8(a), where we are joining the readings from light, temperature, and noise sensors.

There are several disadvantages of computing joins in this manner. First, intermediate results must be materialized in the hash indices of "internal" joins in each plan. Even with left-deep plans which join n sources, $n-2$ additional in-flight indices are needed for intermediate results. For example, in Figure 8(a), intermediate tuples composed of light and temperature readings are stored in the left hash index in the topmost join. We call these *intermediate indices*.

Second, this scheme does not scale well with the number of user queries. For example, imagine we have n sources and one query for each possible 3-way join over the sources. Then there are at least $\binom{n}{3} \sim O(n^3)$ queries requiring one intermediate index. In this example, an intermediate index can be shared among at most $O(n)$ other queries. Thus, we need to maintain at least $\binom{n}{2} \sim O(n^2)$ intermediate indices to support pipelined joins for all the queries. This can be a significant amount of state. Consider an example with 15 distinct types of sensors. We would need to maintain $\binom{15}{2} \sim 105$ intermediate indices to support our hypothetical queries. Further, imagine each tuple is 128 bytes, for each join every tuple matches exactly one other tuple, the sensors produce a tuple every second, and the indices retain the last hour of readings. Then each index on a single stream would be 0.46 MB, and each intermediate index would be 0.9MB. Just to support $\binom{15}{3} \sim 455$ distinct queries, the total size for all in-flight indices would be 101MB.

Third, pipelined joins arranged in a query plan do not permit fine-grain adaptivity of the form offered by the eddy. Every time the join order changes, we must recompute some intermediate indices. In our CACQ system, we avoid these problems by promoting transient indices on individual sources to first class operators, called SteMs, and place them into an eddy. The cascade of doubly-pipelined joins in our example in Figure 8(a) would be converted to the a plan in Figure 8(b) in the eddy framework. The idea of SteMs was first introduced in [28].

SteMs in the CACQ system are simply operators that encapsulate a single index built on a stream using a particular attribute as the key. These indices can be hash indices to support equality joins, which arise in our building monitoring scenario. Or they can be other like B-trees, which support range predicates. SteMs can be passed tuples that are inserted (or *built*) into the index, or tuples which are used to search (or *probe*) the index. A SteM will return all tuples passed to it by the eddy back to the eddy. The `ready` and `done` bits are marked to indicate the operators to which the tuple still needs to be sent. In addition, an *intermediate* tuple that is the concatenation of the tuple used to probe the index and the match, is output for each match (also marked appropriately).

A multiway join is computed by starting with a new tuple pushed into the eddy from some source, a *singleton* tuple, and routing it through SteMs to produce the joined result(s). For example, imagine a query which ranges over all three sources in Figure 8(b). When a new light tuple arrives, one possible route is that it first is inserted into the light SteM. Then it is sent to probe the temperature SteM and joined with a temperature reading. Then the intermediate tuple is sent to the noise SteM, joined with noise readings, and resulting tuple is then output. For a query which ranges over only the light and temperature sources, the eddy can output the intermediate tuple produced after the probe into the temperature SteM. Note that a tuple used to probe SteMs can be either a singleton or an intermediate tuple. Thus the SteM can apply any predicate containing its indexed source and attribute. The eddy routes these tuples by obeying some constraints for correctness, and following a routing policy for efficiency.

Because we have interposed an eddy between the indices, we have lost the atomic "build then probe" property of pipelined joins, leading to two constraints on the eddy and SteMs to ensure correctness. The first constraint is that a singleton tuple must be inserted into all its associated SteMs before it is routed to any of the other SteMs with which it needs to be joined. When it is inserted, it is tagged with a globally unique sequence number. Thus, SteMs only index singleton tuples. The second constraint is that an intermediate tuple returned from a SteM is valid only if the sequence number of the tuple used to probe the SteM is greater than (i.e. it arrived later) the sequence number of the indexed tuple. All valid intermediate tuples retain the larger of the two sequence numbers, and invalid tuples are discarded by the SteM. These constraints maintain the "build then probe" property between any two tuples that are joined, and are sufficient to prevent duplicate tuples from arising. Within these constraints, the eddy is free to choose the order in which to route a tuple to accumulate results. These routing decisions are discussed next.

There are several advantages to using SteMs with an eddy for join processing. First, only a single SteM is built for each source, and these SteMs are shared across all the joins among all the queries posed. Contrast the scalability of this scheme with the scalability of pipelined joins in a tree. Using our previous example with 15 sensors, SteMs would only need to maintain 6.9 MB of data to support any subset of the possible 32K (2^{15}) joins, compared with 101 MB to support only 455 queries. Second, we can compute joins in a pipelined fashion for all possible joins over the sources. Third, the join order is decided on a per-tuple basis, providing fine-grain adaptivity.

3.3.2 Routing with Joins

Routing tuples in an eddy in our CACQ system involves two computations. The first is to determine to which set of operators a tuple can be sent next or to which queries it can be output. The second is to choose among the possible operators to which the tuple can go. For the second decision, we use the same routing scheme as the one used for selection operators to route tuples to SteMs. For the first computation, we need to maintain additional data-structures and augment our current ones to handle generating and routing intermediate tuples.

First, we need to augment the state associated with each source. We add a separate SteMs list containing SteM operators with which the source needs to be joined. The query list remains the same; the list includes the queries that contain *only* that source. Thus, the masks in the `completionMask` list are padded with 0s for each SteM in the SteMs list. Similarly, we augment the `ready` and `done` bits in the tuple state to include bits for the added SteMs. These changes provide a scheme for routing singleton tuples into SteMs; we just need to add data-structures that handle intermediate tuples.

Intermediate tuples can contain data from a some subset of sources flowing into the eddy. Given k input sources, there are at most 2^k possible types of tuples. Analogous to the state we maintain for existing sources, we create a virtual source, with a unique `sourceId`, for each possible type of intermediate tuple. Thus, each source or virtual source is associated with some distinct subset of the sources in the system. With each virtual source, we associate an operators list, SteMs list, query list, and `completionMask` list. All queries which range over all the sources of a virtual source are in the queries list corresponding to that virtual source. The operators list is the union of all selection operators that need to be applied for each query in the queries list. The SteMs list contains all the SteMs modules with which an intermediate tuple needs to be joined to satisfy queries which range over additional sources. The `completionMask` list contains a bit mask for each query. Likewise, each `completionMask` indicates which operators in the operators list need to be applied on a tuple to output or reject the tuple for that query.

When an intermediate tuple is formed, its `queriesCompleted` bitmap is cleared and is tagged with the `sour-ceId` of its new virtual source. The `ready` bits are set to reflect the operators in the operators and SteMs list to which the tuple still can go to next. Also, the `done` bits are set to indicates which operators have already been applied to its data. As usual, the eddy compares the `completionMask` to the `done` bits to determine to which queries an intermediate tuple can be output. The eddy uses the `ready` bits to determine to which SteMs and selection operators a tuple can be sent. We omit the details for efficiently performing these bit vector initialization and manipulations due to lack of space.

In our assumptions about the CACQ system, we assumed that the number of sources were limited, and that many queries join the sources in the same way. While the size for maintaining each source's state is usually small, the state for 2^k virtual sources could be too large to handle, so we make an optimization. We only create a virtual source and its associated state for each distinct subset of sources that has a query ranging over it. For intermediate tuples without a virtual source, we retain the `sourceId` from the tuple which was used to probe the SteM. We use the source or virtual source associated with that `sourceId` to route the intermediate tuple. Hence we limit the number of virtual sources to at most the number of queries.

When a new query arrives into the system, first it is added to the queries list of the virtual source corresponding to the sources over which the query ranges. If a virtual source does not exist, it is created. We determine the selection operators and the SteMs that the query will need. The selection operators are folded into the system as described in Section 3.2. SteMs are treated no differently than selection operators. If new a SteM is added, then that SteM is added to the SteMs list for all existing sources and virtual sources which contain the source associated with the SteM.

3.3.3 Purging SteMs

Because our CACQ system is designed to operate over streams, a mechanism is needed to limit the state that accumulates in joins as streams flow endlessly into them. This mechanism, proposed in [33], is to limit the number of tuples in a particular SteM by imposing a window on the stream. Windows specify a particular number of tuples or period of time to which the join applies; tuples outside the window are not included in the join. Thus, they are a key component of our join solution, although, from a research perspective, they have been thoroughly discussed.

We allow windows to be specified as a component of a join predicate. In the current implementation, windows are simply a fixed number of tuples; extending the system to allow windows over a fixed time period would be fairly simple. Windows are *sliding*: the window always incorporates the most recent data in the stream. As new tuples flow in, old tuples are forced out of the window. [1]

Since SteMs may contain multiple predicates, we cannot simply discard tuples from the index that do not fall within the window of a particular predicate. We keep the maximum number of tuples specified among all the windows associated with the predicates. For a given predicate, we reject matches that are outside of that predicates window but still within the index. In this way, we do not have to create multiple SteMs to support different window sizes.

Now that we have seen the space savings SteMs can provide, we are finished with the architectural elements of the system. In the next Section we discuss an important design point: building a routing policy to efficiently route tuples between operators in a continuous eddy.

[1]There are other possible models for windows, such as a jumping window where the window does not move until a window's worth of new tuples have arrived. For a discussion of such issues, see [11, 33, 35, 26].

4 Routing Policies

The routing policy is responsible for choosing which tuple should be routed next and which operator it should be routed to. The original eddy implementation used two ideas for routing: the first, called *back-pressure*, limits the size of the input queues of operators, capping the rate at which the eddy can route tuples to slow operators. This causes more tuples to be routed to fast operators early in query execution, which is intuitively a good idea, since those fast operators will filter out some tuples before they reach the slower operators. The second approach augments back-pressure with a ticket scheme, whereby the eddy gives a ticket to an operator whenever it consumes a tuple and takes a ticket away whenever it sends a tuple back to the eddy. In this way, higher selectivity operators accumulate more tickets. When choosing an operator to which a new tuple should be routed, the ticket-routing policy conducts a lottery between the operators, with the chances of a particular operator winning proportional the number of tickets it owns. Thus, higher selectivity operators will receive more tuples early in their path through the eddy.

We have implemented a variant of the ticket scheme. In our variant, a grouped-filter or SteM is given a number of tickets equal to the number of predicates it applies, and penalized a number of tickets equal to the number of predicates it applies when it returns a tuple back to the eddy. A SteM that outputs more tuples than it receives could thus accumulate negative tuples; we lower bound the number of tuples any modules receives at one. Multiple SteMs with only one ticket will be scheduled via back-pressure, since higher cardinality joins (which should be scheduled towards the top of the plan) will require longer to completely process each input tuple. Thus, highly selective grouped filters will receive more tickets, and tuples will be routed to these filters earlier in processing. In this way, we favor low-selectivity via tickets and quick work via backpressure. We weight the value of that work by the number of predicates applied by each operator.

We now present a performance evaluation of our modified ticket-based routing scheme as it functions with a number of selection-only queries. We will discuss the performance of our routing policy with respect to joins as a part of the experiments in Section below.

4.1 Ticket Based Routing Studies

The modified ticket-based routing scheme presented above is designed to order filter-operators such that the most selective grouped filter that applies to the most predicates is applied first.

We compare this scheme to three alternatives. In the *random* scheme, tuples are routed to a random operator that they have not previously visited. In the *optimal* scheme, tuples are routed to the minimum set of filters required to process the tuple. This is a hypothetical scheme that provides an upper bound on the quality of a routing scheme. For any given tuple, it applies the smallest number of possible filters.

The optimal approach orders selections from most to least selective, and always applies them in that order. Determining this optimal ordering is not always possible, since the underlying distribution of an attribute may be unknown or not closely match any statistics gathered for that attribute. However, for the workload shown in Table 3, clearly the optimal ordering places first applies the selection over $S.a$, then the selection over $S.b$, then $S.c$, $S.d$, and $S.e$. All of the tuples will pass through the $S.a$ selection. Only ten-percent of the tuples will pass $S.a$, thirty percent of those will pass $S.b$, and so on. This leads to the following expression for the expected number of filters each tuple will enter in this approach:

$1+$; All tuples apply S.a filter
$(1-0.9)+$; Tuples that apply S.b filter
$(1-0.9)\times(1-0.7)+$; S.c filter
$(1-0.9)\times(1-0.7)\times(1-0.5)+$; S.d filter
$(1-0.9)\times(1-0.7)\times(1-0.5)\times(1-0.3)$; S.e filter
$= 1.15$

Table 3: **Queries for Routing Scheme Comparison .**

```
1. from S select S.index where S.a > 90
2. from S select S.index where S.a > 90 and S.b > 70
3. from S select S.index where S.a > 90 and S.b > 70 and
         S.c > 50
4. from S select S.index where S.a > 90 and S.b > 70 and
         and S.c > 50 and S.d > 30
5. from S select S.index where S.a > 90 and and S.b > 70 and
         S.c > 50 and S.d > 30 and S.e > 10
```

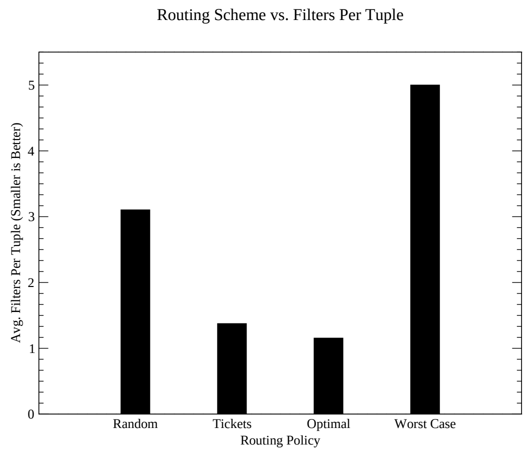


Figure 9: *Comparison of Various Routing Schemes*

We do not expect any routing scheme to perform this well, but it serves as a useful lower bound on the number of filters that must be applied.

The final alternative is a hypothetical *worst-case* approach, in which every filter is applied to every query: 5 filters, in the workload shown below. No routing scheme should perform this badly.

We ran experiments to show how the ticket based scheme compares to these other approaches for the fixed queries shown in Table 3. We chose to use a fixed set of queries rather than random queries because queries with predicates over a uniformly selected random range will tend to experience little overlap and all select about the same number of tuples, causing the random and ticket schemes to perform similarly. Since the goal of this experiment is to show that the ticket-based scheme can effectively determine selectivities of grouped filters when that affects performance, we felt this was an appropriate decision.

We ran the system for one minute and compared the total number of tuples scanned to the tuples entering each filter operator. Figure 9 shows the results: our ticket-based routing scheme routes the average tuple to just 1.3 filters, while the randomized scheme routes every tuple to about 3.2 filters.

4.2 Adapting to Changing Workloads

In addition to routing tuples efficiently, one of the properties of our continuous query system is that it can rapidly adapt to changing workloads. To demonstrate this, we ran experiments with three query workloads, as shown in Table 4. Queries are over the same source S as in the previous experiments. In these experiments, the first query was introduced at time 0, and each successive query was introduced five seconds later. In the first workload, queries are independent, and so, just as with a conventional eddy, the most selective predicates should be executed first since those are the most likely to filter out tuples. In this case, query five is the most selective. The second workload shows the capability of the ticket-based scheme to prioritize filters that apply to different numbers of predicates: all filters have the same selectivity, but five times as many predicates are applied to S.a as S.e. The final workload is much more complex: queries share work and have filters with a range of selectivities. The correct ordering of these queries is not immediately apparent.

Figure 10 shows the percentage of tickets routed to each filter over time for the three workloads. Percentage of tickets received is a measure of the routing policy's perceived value of an operator. Highly selective operators are of higher value because they reduce the number of tuples in the system, as are operators which apply predicates for many queries, because they perform more net work. Before a filter is introduced it receives zero tickets; notice how quickly the system adapts to newly introduced filters: in most cases, four seconds after a filter is added the percentage of tuples it receives has reached steady-state.

Workload 1 and 2 settle to the expected state, with the most selective, most frequently applied filters receiving the bulk of the tickets. Workload 3 has results similar to workload 2, except that the S.a, S.b, and S.c filters all receive about the same name number of tickets once all queries have been introduced. This is consistent, because S.b and S.c are more selective, but apply to fewer queries so are weighted less heavily. Also note that S.d and S.e receive slightly more tickets than in Workload 2; this is due to the increased selectivity of their predicates.

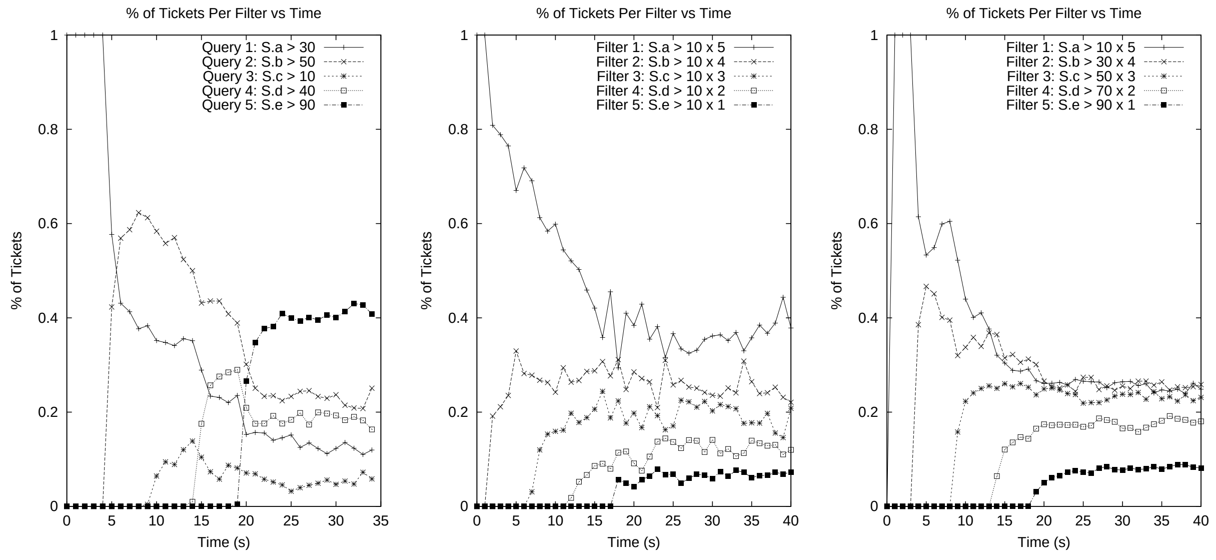
5 Performance Study: Comparing with Other Continuous Query Systems

To demonstrate the effectiveness of our CACQ system, we compare it with the approach used by the recently published NiagaraCQ system [8, 7]. NiagaraCQ uses a static query optimizer to build fixed query plans for continuous queries. NiagaraCQ's plans are *grouped*, which means that operators are shared between queries when possible. The optimizer allows two queries to share an operator if it can demonstrate that the set of tuples flowing into that operator in both queries is always the same. This "identical tuple sets" requirement must hold because tuples have no explicitly encoded lineage, as in our CACQ approach, so the queries to which a tuple may be output can only be inferred from the tuple's location in the query plan. In practice, this means that very little overlap will be possible between queries of any complexity: although it may be possible to share an initial selection, any operators which follow that selection must be replicated across all queries (even if they have exactly the same predicates), because the tuples flowing into those operators are not identical.

Rather than creating a predicate index for selection operators, NiagaraCQ combines selections over an attribute into a join between the attribute and the constants from the selection predicates. Because an efficient join algorithm

Table 4: **Query Workloads for Adaptivity Scenario.**

Workload 1	
1.	`select S.index from S where S.a > 30`
2.	`select S.index from S where S.b > 50`
3.	`select S.index from S where S.c > 10`
4.	`select S.index from S where S.d > 40`
5.	`select S.index from S where S.e > 90`
Workload 2	
1.	`select S.index from S where S.a > 10`
2.	`select S.index from S where S.a > 10 and S.b > 10`
3.	`select S.index from S where S.a > 10 and S.b > 10 and S.c > 10`
4.	`select S.index from S where S.a > 10 and S.b > 10 and and S.c > 10 and S.d > 10`
5.	`select S.index from S where S.a > 10 and and S.b > 10 and S.c > 10 and S.d > 10 and S.e > 10`
Workload 3	
1.	`select S.index from S where S.a > 10`
2.	`select S.index from S where S.a > 10 and S.b > 30`
3.	`select S.index from S where S.a > 10 and S.b > 30 and S.c > 50`
4.	`select S.index from S where S.a > 10 and S.b > 30 and and S.c > 50 and S.d > 70`
5.	`select S.index from S where S.a > 10 and and S.b > 30 and S.c > 50 and S.d > 70 and S.e > 90`



(a) Workload 1 (b) Workload 2 (c) Workload 3

Figure 10: *Percentage of Tickets Routed to Filters Over Time*. Notice that the most selective predicates (S.e $>$ 90) in (a) and (S.a $>$ 10) in (b) rapidly adapt to receive the most tickets, which are correlated with their routing priority in the eddy.

can be used if a B-Tree index is built on the predicates, this approach is similar in efficiency to our predicate index. However, when predicates overlap, multiple copies of every tuple are produced as output of the NiagaraCQ join, which imposes a non-trivial performance overhead.

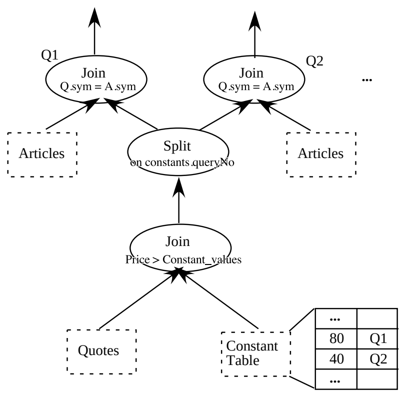
To compare the two systems, we run experiments like those proposed in [7]. In these experiments, we execute queries of the form:

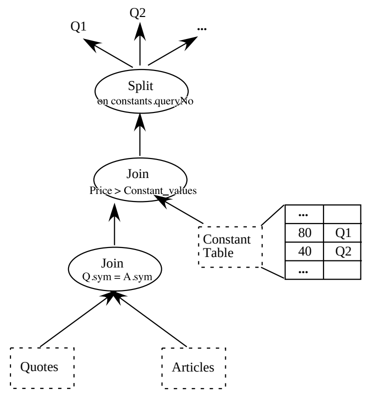
```
select * from stocks, articles where stocks.price > x and stocks.symbol = articles.symbol
```

`Stocks` is a list of stock quotes, and `articles` is a set of news articles about the companies in those quotes. Articles ranged from about 200 bytes to 1 kilobyte. Stock prices were randomly selected using a uniform distribution over the domain (0,100]. We run a number of queries of this form, varying only the value of `x`. Notice that this workload is very favorable towards the NiagaraCQ approach, because there is complete overlap between queries. A more mixed assortment of queries would make it much harder for the NiagaraCQ optimizer to perform its grouping.

The NiagaraCQ optimizer generates two possible plans for these queries, which consist of a selection operator and a join. In the first, called *PushDown* (Figure 11(a)) the selection operator is placed below the join in the query plan. All selections can be placed in the same group, because they are all over the unfiltered `stocks` relation. However, the join operators cannot be placed into the same group because the sets of tuples from each query's selection are disjoint – a separate join must be run for each query (although the hash table over `articles` is shared between queries.) The *split* operator shown in the plan is a special operator that divides the output of a grouped operator based on the query specified by the `file` attribute of the constants table.

The other alternative, called *PullUp*, shown in Figure 11(b) places the join at the bottom of the query plan. Since the tuples flowing in from both relations are unfiltered in all queries, every join operator can be placed in a single group. Since all queries use exactly the same join predicate, the output of the grouped join into every selection predicate is identical. Thus, those selection predicates can all be placed into a single group. As the results in Figure 12 show, this





(a) PushDown Plan (b) PullUp Plan

Figure 11: *Two Alternative Query Plans in NiagaraCQ*

PullUp approach is (not surprisingly) more efficient because there are not many copies of the join operator. Notice, however, that it suffers from a disturbing problem – the selection predicates must be applied after the join, which is contrary to well establish query optimization wisdom [31].

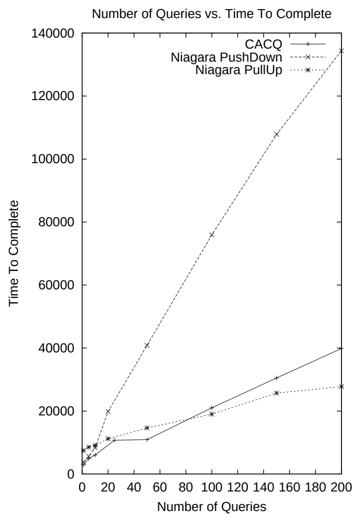
We compared these two alternatives to the same query in our CACQ system. In our system, this query consists of three operators: a grouped filter on `stocks.price`, and a pair of SteMs on `stocks.symbol` and `arti-cles.symbol`. We used the modified ticket-based routing scheme discussed above to schedule tuples between the SteMs and the grouped filter.

We manually constructed NiagaraCQ query plans in Telegraph with the structure shown in Figure 11. When emulating NiagaraCQ, we removed the per-tuple data structures and the code that manages them, since these are specific to our CACQ approach. Notice that we also did not include materialization operators in the NiagaraCQ plans, as was done in that work, since we were able to keep all tuples and join tables in main memory. As was done in [7], we ran experiments that delivered a fixed size update to stock prices and news articles (2250 stocks and articles, with one article per stock). Query selection predicates were randomly chosen from the uniform distribution, although we insured that the union of all predicates selected 100% of the tuples. We placed no limit on the sum of selectivities (as was done in the NiagaraCQ work), because doing so does not significantly affect the performance of their best approach or our system.

We varied the number of distinct queries (distinct selection predicates) from 1 to 200 and compared the performance, show in in Figure 12. Notice that the CACQ approach is faster than the PullUp approach for small numbers of queries, but that it later becomes slightly slower. The original performance benefit is because CACQ can apply selections on smaller tuples before it computes the join; the PullUp approach joins all tuples first. For large numbers of queries, the CACQ approach is somewhat slower due to the additional time spent maintaining the tuple state (which was not included in the NiagaraCQ experiments). The PushDown approach (as was the case in the NiagaraCQ experiments) is slower in all cases.

Note that the shape of the lines for the two NiagaraCQ experiments shown in Figure 12 closely matches the shape of the lines shown in Figure 4.3 of [7], suggesting that our emulation of the NiagaraCQ approach is sound. These experiments show that our CACQ approach is capable of matching the best of the NiagaraCQ approaches without the benefit of a cost-based query optimizer. Our routing policy determines that the grouped selection on prices is more selective than the join, and thus routes tuple through the selection first.

In the next set of experiments, we modify the above scenario to apply a UDF over batches of stock quotes which



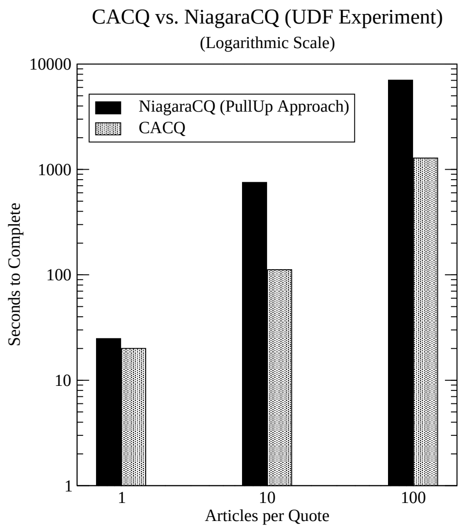


Figure 12: *Comparing the Performance of the two NiagaraCQ approaches to CACQ*

Figure 13: *NiagaraCQ PullUp Scheme vs. CACQ with UDF Selections*

flow into the system. We fix the number of simultaneous queries at 100, but we vary the number of articles per stock quote, to simulate multiple news stories and sources reporting about a particular company. In this modified approach, each user specifies a UDF that selects stock quotes of interest (instead of a single $>$ predicate). Quotes are shipped in batches reflecting several hours worth of activity, to allow UDFs to utilize more than just the most recent stock price in deciding to return a particular quote. This is the sort of environment serious investors might use: each user's UDF would search for the parameters he thought were particularly important in determining when to buy or sell a stock; when those parameters are satisfied, quotes and recent news articles about those quotes are returned. Notice that in this case, we cannot use a grouped filter to evaluate UDFs and NiagaraCQ cannot group UDFs via a BTree. The results of these experiments are shown in figure 13. We compared only against the PullUp approach, as the PushDown approach remains much slower than CACQ, for the same reasons as in the previous experiment. We varied the cardinality of the `articles` relation so that there were 1, 10, or 100 articles per quote. We simulated the cost of a UDF by spin-looping for a randomly, uniformly selected time over the interval of 10 - 500 μS.

In this case, the CACQ approach is much more efficient because CACQ applies UDFs to stock quotes before they are joined with articles, while NiagaraCQ must apply UDFs after the join if it wishes perform only a single join. As the cardinality of `articles` increases the expensive UDFs must be applied many more times in the NiagaraCQ approach than in CACQ. In general, this is a limitation of the NiagaraCQ approach which cannot be overcome unless lineages are explicitly encoded. NiagaraCQ cannot push selections from two queries below a shared join without performing the join multiple times; if the fanout of the join is much greater than one, this will severely impair NiagaraCQ's performance. Furthermore, as we saw in the single-article case, NiagaraCQ pays a penalty for performing selections on larger, joined tuples.

6 Related Work

The integration of Eddies and continuous queries in our CACQ system is necessarily related to both areas of research. We summarize this research here, and also discuss related systems for adaptive query processing, stream processing, and query processing over sensor data.

Eddies were originally proposed in [4]. The basic query operator and the back-pressure and ticket-based routing schemes were developed. The work was not specifically focused on continuous query processing, although, as we have discussed, attention to adaptivity and responsiveness in eddy lead to some shared attributes with continuous query systems, such as the use of non-blocking, pipelined operators and the ability to perform mid-query reoptimization via tuple lineage. Our single query, single source query processing scheme is essentially the same as that proposed in the original eddy work, and our ticket-based routing scheme is a simple adaption of the scheme proposed in [4].

Notions of adaptivity and pipelining are well established in the research community. Parallel-pipelined joins were proposed in [40]. Adaptive systems such as XJoin, Query Scrambling, and Tukwila [38, 39, 18] demonstrated the importance of pipelined operators to adaptivity.

6.1 Continuous Queries

Existing work on continuous queries provides techniques for simultaneously processing many queries over a variety of data sources. These systems propose the basic continuous query framework that we adopt and also offer some extensions for combining related operators within query plans to increase efficiency. Generally speaking, the techniques employed for doing this combination are considerably more complex and less effective at adapting to rapidly changing query environments than the CACQ system we have proposed.

Efficient trigger systems, such as the TriggerMan system[13] are similar to continuous queries in that they perform incremental computation as tuples arrive. In general, the approaches used by these systems is to use a discrimination network, such as RETE [12] or TREAT [24], to efficiently determine the set of triggers to fire when a new tuple arrives. These approaches typically materialize intermediate results to reduce the amount of work required for each update.

Continuous queries were proposed and defined in [37] for filtering of documents via a limited, SQL-like language. In the OpenCQ system [22], continuous queries are likened to trigger systems where queries consists of four element tuples: a SQL-style query, a trigger-condition, a start-condition, and an end-condition.

The NiagaraCQ project [8] is the most recently described CQ system. Its goal is to efficiently evaluate continuous queries over changing data, typically web-sites that are periodically updated, such as news or stock quote servers. Users install queries that consist of an XML-QL [10] query as well as a duration and re-evaluation interval. Examples of the NiagaraCQ grouping approach are given in Section 5 above. The NiagaraCQ approach to optimizing continuous queries is substantially different from the approach presented in this paper: grouping is performed by a static query optimizer, and the order in which operators are applied is fixed by this query plan. Although plans can be "dynamically regrouped", heuristics for when to do so are unclear, and, unlike our continuous adaptivity, regrouping is an expensive operation that cannot be performed frequently. Furthermore, NiagaraCQ does not explicitly encode lineage into tuples, and so cannot group related operators in two different queries if the sets of tuples flowing into those operators are not identical.

The XFilter system [2] is another recent continuous-query system. XML-documents stream into the system and are matched to filter-profiles expressed in the XPath [9] language. It optimizes queries by indexing profiles based on the filter conditions that appear within those profiles. Thus, when a new XML document arrives in the system, each of its tags is matched against this filter-condition index to rapidly determine which profiles have conditions that need to be checked against the document: in our terms, it is an extremely efficient grouped-filter for matching XML-documents and XPath predicates. It is explicitly focused on handling the structure of XML and does not address joins or adaptivity.

[5] is a "vision" paper presenting many high-level goals for continuous query systems over streams. It notes the need for adaptivity, pipelined and windowed operators, grouping, and efficient sharing of data structures.

The problem of sharing working between queries is not new. Multi-query optimization, as discussed in [32] seeks to exhaustively find an optimal query plan, including common subexpression, between a small number of queries. Recent work, such as [30, 25] provides heuristics for reducing the search space, but is still fundamentally based on the notion of building a *query-plan*, which we avoid in this work.

6.2 Streams

Fundamental notions of stream processing are presented in [33], including extensions to SQL for windows and discussions of non-blocking and timestamped operators. [11] proposes windows as a means of managing joins over very large sets of data. [35] discusses operators for processing streams in the context of network routing; it includes an interesting discussion of appropriate query languages for streaming data.

[27] discusses models of data streaming from sensors. [23] proposes using continuous queries for processing over streams of sensor data and offers motivating performance examples, but falls short of providing a specific framework for query evaluation and does not incorporate adaptivity.

7 Future Work and Conclusions

In this paper we present the first continuous query implementation based on a continuously adaptive query processing scheme. We show that our eddy-based design provides significant performance benefits, not only because of its adaptivity, but also because of the aggressive cross-query sharing of work and space that it enables. By breaking the abstraction of shared relational algebra expressions, our Telegraph CACQ implementation is able to share physical operators – both selections and join state – at a very fine grain. We augment these features with a grouped-filter index to simultaneously evaluate multiple selection predicates.

Our approach provides significant performance and scalability benefits over prior continuous query architectures, both in terms of raw performance and also robustness to changes in the system workload. The CACQ architecture also raises a number of opportunities for future research. First, we have focused to date on the relational queries natural in sensor networks – sensors provide simple, structured “reading” vectors, which are direct analogs of relational tuples. However there has been interest in recent years in continuous queries over richer XML documents, and we believe our CACQ approach will be beneficial in that arena as well. A promising direction is to integrate our CACQ work with a grouped filter appropriate to XML queries – for example, the XFilter technique of [2]. A second arena for future work is to explore more sophisticated eddy routing policies. The adaptivity of Eddies relates them closely to work in control theory [3, 6, 1, 20, 29] and reinforcement learning [36], and we are interested in extending eddy routing policies to adaptively learn data dependencies and query dependencies in a more sophisticated manner than typically done in database query optimizers. An analysis of a broad set of policies from both a theoretical and an empirical approach would be beneficial. A third area of future work has to do with sharing the computation of aggregates between queries – maintenance of aggregate groups can require substantial storage, indicating that sharing those groups between related queries is a fruitful area of research. A fourth interesting direction is to explore a tighter integration of Telegraph with sensors in a sensor network. In many multi-hop sensor networks, it is critical from a power-savings and performance perspective to minimize communication in the network [17]. This requires pushing query operators like selections, aggregates and even eddies to the sensors themselves, enabling them to run extremely wide-scale distributed continuous queries. This is a rich direction for future investigation.

References

[1] R. Agrawal. Sample mean based index policies with $o(\log n)$ regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27:1054–1078, 1995.

[2] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *International Conference on Very Large Data Bases*, September 2000.

[3] P. Auer. Using upper confidence bounds for online learning. In *The 41st Annual Symposium on Foundations of Computer Science*, Nov. 2000.

[4] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM SIGMOD*, pages 261–272, Dallas, TX, May 2000.

[5] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, September 2001.

[6] D. A. Berry and B. Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall, 1985.

[7] J. Chen, D. DeWitt, and J. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *To appear: International Conference on Data Engineering*, San Jose, CA, February 2002.

[8] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the ACM SIGMOD*, 2000.

[9] J. Clark and S. DeRose. XML path language (XPath) version 1.0, November 1999. http://www.w3.org/TR/xpath.

[10] A. Deutsch, M. Fernandez, D. Floresc, A. L. , and D. Suciu. XML-QL: A query language for XML, 1998. http://www.w3.org/TR/NOTE-xml-ql.

[11] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proceedings of the 17th Conference on Very Large Databases*, Barcelona, Spain, 1991.

[12] C. Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[13] E. Hanson, N. A. Fayoumi, C. Carnes, M. Kandil, H. Liu, M. Lu, J. Park, and A. Vernon. TriggerMan: An Asynchronous Trigger Processor as an Extension to an Object-Relational DBMS. Technical Report 97-024, University of Florida, December 1997.

[14] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of Fifth ACM Conf. on Mobile Computing and Networking (MOBICOM))*, Seattle, WA, August 1999.

[15] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, and D. C. K. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[17] C. Intanagonwiwat, R. Govindan, , and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *In Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM 2000)*, Boston, MA, August 2000.

[18] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of the ACM SIGMOD*, 1999.

[19] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Mobile networking for smart dust. In *Proceedings of Fifth ACM Conf. on Mobile Computing and Networking (MOBICOM))*, Seattle, WA, August 1999.

[20] T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.

[21] N. Lanham. The telegraph screen scraper, 2000. http://db.cs.berkeley.edu/ nickl/tess.

[22] L. Liu, C. Pu, and W. Tang. Continual queries for internet-scale event-driven information delivery. *IEEE Knowledge and Data Engineering*, 1999. Special Issue on Web Technology.

[23] S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. San Jose, CA, February 2002. To Appear, International Conference on Data Engineering, 2002.

[24] D. P. Miranker. Treat: A better match algorithm for ai production system matching. In *Proceedings of AAAI*, pages 42–47, 1987.

[25] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, 2001.

[26] C. Nippl, R. Rantzau, and B. Mitschang. Streamjoin: a generic database approach to support the class of stream-oriented applications. In *Database Engineering and Applications Symposium*, pages 83–91, 2000.

[27] P.Bonnet, J.Gehrke, and P.Seshadri. Towards sensor database systems. In *2nd International Conference on Mobile Data Management, Hong Kong*, January 2001.

[28] V. Raman. *Interactive Query Processing*. PhD thesis, UC Berkeley, 2001.

[29] H. Robbins. Some aspects of the sequential design of experiments. *Bulletin American Mathematical Society*, 55:527–535, 1952.

[30] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 249–260, 2000.

[31] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. pages 23–34, Boston, MA, 1979.

[32] T. Sellis. Multiple query optimization. *ACM Transactions on Database Systems*, 1986.

[33] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database systems. In *International Conference on Very Large Data Bases*, Mumbai, India, September 1996.

[34] M. Shah, S. Madden, M. Franklin, and J. M. Hellerstein. Java support for data intensive systems. *SIGMOD Record*, December 2001. To Appear.

[35] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, LA, June 1998.

[36] R. S. Sutton and A. G. Bartow. *Reinforcement Learning*. MIT Press, Cambridge, MA, 1998.

[37] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 321–330, 1992.

[38] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, pages 27–33, 2000 2000.

[39] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD*, 1998.

[40] A. Wilschut and P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the International Conference on Parallel and Distributed Information Systems (PDIS*, pages 68–77, December 1991.