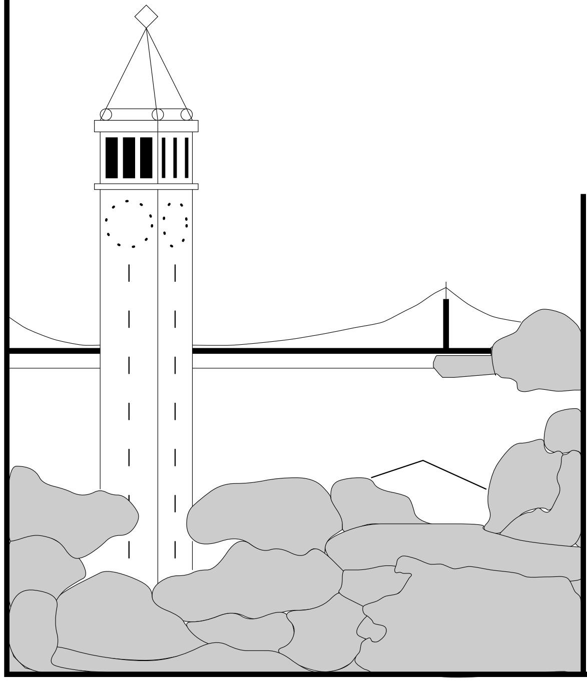
# Flux: An Adaptive Partitioning Operator for Continuous Query Systems

*Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran*
*and Michael J. Franklin*
*CS Division, EECS Department, U.C. Berkeley*
{*mashah, jmh, sirish, franklin*}*@cs.berkeley.edu*



**Report No. UCB/CSD-2-1205**

October 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This work was supported by NSF grant nos. 0122599, 0208588, a UC MICRO grant, and gifts from Microsoft, IBM, and Intel. Infrastructure for the research was provided by NSF grant EIA-9802069.

# Flux: An Adaptive Partitioning Operator for Continuous Query Systems

Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran and Michael J. Franklin
CS Division, EECS Department, U.C. Berkeley
{mashah, jmh, sirish, franklin}@cs.berkeley.edu

## Abstract

*The long-running nature of continuous queries poses new scalability challenges for dataflow processing. CQ systems execute pipelined dataflows that may be shared across multiple queries. The scalability of these dataflows is limited by their constituent, stateful operators – e.g. windowed joins or grouping operators. To scale such operators, a natural solution is to partition them across a shared-nothing platform. But in the CQ context, traditional, static techniques for partitioned parallelism can exhibit detrimental imbalances as workload and runtime conditions evolve. Long-running CQ dataflows must continue to function robustly in the face of these imbalances.*

*To address this challenge, we introduce a dataflow operator called* Flux *that encapsulates adaptive state partitioning and dataflow routing. Flux is placed between producer-consumer stages in a dataflow pipeline to repartition stateful operators while the pipeline is still executing. We present the Flux architecture, along with repartitioning policies that can be used for CQ operators under shifting processing and memory loads. We show that the Flux mechanism and these policies can provide several factors improvement in throughput and orders of magnitude improvement in average latency over the static case.*

## 1. Introduction

Continuous query (CQ) systems break the traditional paradigm of a request-response style system in which a finite answer set is computed for each query. Instead, users register queries that specify their interests over unbounded, streaming data sources. Based on these queries, a query engine continuously filters and synthesizes incoming data from the sources, and delivers unbounded, streaming results to the appropriate users. These systems have been proposed for a variety of personalization and critical monitoring tasks. For example, they can be used to deliver up-to-date personalized news, monitor stock quotes, filter incoming email for spam, monitor traffic for accidents and congestion, or monitor the network for troubleshooting and intrusion detection.

These demanding applications place several scalability requirements on continuous query systems. First, a typical service could have millions of users each with several queries registered. Second, results need to be computed from incoming data upon arrival, so fast response times and high data throughput are essential. Finally, since queries range over endless streams of data, a CQ system may need to manage data over a large history to process these queries. Previous approaches have addressed these scalability issues in two ways. One thrust has been to leverage the commonality amongst queries [6, 16, 7, 2] and index queries like data in a traditional database system. Another direction has been to exploit weaker semantics and sacrifice result quality for cases in which the system cannot be scaled to match peak workloads [18, 4]. While this previous work has focused on single-site implementations, we present complementary techniques for parallelizing CQ dataflows that can offer increased scalability to any CQ system.

### 1.1. Parallelism and Adaptive Partitioning

Pipelined CQ dataflows are fundamentally limited by the scalability of their constituent streaming, lookup-based operators. The traditional approach for scaling operators is to partition them across a shared-nothing cluster. Often referred to as intra-operator or partitioned parallelism, this approach allows lookup-based CQ operators like windowed joins and group-by-aggregate, which may easily outgrow a single-site's main memory, to utilize aggregate main memory and other resources. Shared nothing clusters can scale to thousands of computers, scaling available main memory, processors, disk space and bandwidth along the way[8, 28, 30], and thereby provide the potential for high throughput and low latencies. Yet, to date, the shared-nothing approach has been overlooked for CQ systems.

CQ systems strain traditional dataflow parallelism techniques, because they require *adaptive, online repartitioning* of lookup-based operators. This requirement arises in CQ systems for two reasons. First, CQ systems typically handle multi-user workloads that place unpredictable demands on resources. As additional queries arrive, new dataflows may be instantiated, causing variations in load and memory usage to arise across a cluster. Second, since by definition continuous queries run indefinitely, CQ operators will encounter changes in system and workload properties in the middle of ongoing queries. For such queries to continue running efficiently, their constituent dataflow operators must be able to adapt *on the fly* to changes in the runtime environment. For lookup-based operators, load and memory usage at each site in a cluster is largely determined by the data partitioning. Thus, the system must adjust intra-operator data partitioning on the fly to balance resource utilization, and hence optimize performance.

### 1.2. Flux

We introduce an encapsulated dataflow operator called *Flux*, Fault-tolerant Load-balancing eXchange. In this paper, we focus on its load-balancing features that adaptively repartition the state of lookup-based operators; we defer the fault-tolerance discussion to future work. Flux is interposed between a partitioned producer-consumer pair in a parallel dataflow pipeline to provide adaptive repartitioning for lookup-based operators on the consumer side. In addition to providing the encapsulated data partitioning found in traditional parallel dataflow systems [11], Flux provides two main adaptivity mechanisms:

1. To mask short-term imbalances, Flux provides a buffering and reordering mechanism that absorbs transient, localized perturbations.

2. To adapt to long-term imbalances, Flux encapsulates the logic for detecting imbalances across a cluster, and for triggering and enabling online repartitioning of state accumulated in lookup-based operators.

This parallel repartitioning mechanism can also be exploited to allow in-core, streaming operators to “spill” to disk, relieving the operator writer of that burden.

In this paper, we present the generic Flux load-balancing mechanism, and also some specific parallel repartitioning policies that handle different causes of imbalance. We present a policy to handle load imbalances during situations with ample memory resources, a policy to handle memory utilization imbalances in a memory-constrained environment, and a hybrid that is effective in both cases. While Flux can be used with a wide range of lookup-based operators, in this paper we focus on its application to operators found in data-stream processing systems like Telegraph [5] and Aurora [4]. In this context, we show that the imbalances outlined above can severely degrade the maximum sustainable throughput and the average response time of statically-partitioned lookup-based operators. By contrast, we show that Flux can achieve several factors improvement in throughput, and orders of magnitude improvement in average response time over the static case.

In the next section, we briefly outline the traditional approach for parallelizing dataflows, its deficiencies, and our contribution. Section 3 describes the methodology used for evaluating Flux. Section 4 describes the Flux buffering and reordering mechanism. Section 5 describes the state movement mechanisms and repartitioning policy to balance load in low memory pressure conditions. Section 6 describes the memory-constrained mechanism and repartitioning policy, and the hybrid policy. Section 7 surveys the related work, and Section 8 summarizes and presents future work.

## 2. Background and Contribution

In this section, we describe previous approaches for optimizing and executing parallel query plans, delineate their inadequacies in the CQ context, and outline our contribution. For this work, we assume a *shared-nothing* [28] or *cluster*-based parallel computing architecture in which each processing node (or site) has a private CPU, memory, and disk, and is connected to all other nodes via a high-bandwidth, low-latency network.

### 2.1. Previous Approach

A popular approach for parallel database queries consists of two phases [13]. First, a query optimizer generates a static, sequential query plan based on fixed cost models and previously computed statistics. Second, at execution time, this plan is parallelized based on the current runtime characteristics of the system. In the second phase, the degree of parallelism (or declustering) of operators is determined, and a corresponding number of *instances* of the operator are created, with each instance responsible for a certain portion or partition of the input data. The sites in a cluster for each of these instances is chosen, and operators are then instantiated to execute the query plan. Except for some constrained cases discussed in Section 7, the partitioning of these operators traditionally remains fixed until the query is completed.

### 2.2. The Problem of Changing Loads

Continuous query systems like [16, 7, 4] are essentially a collection of pipelined dataflows, a generalization of pipelined query plans, that are shared across numerous queries. A dataflow pipeline is a DAG in which the nodes are non-blocking operators, and the edges represent the direction in which data is processed. A *stage* in a dataflow pipeline is a producer-consumer operator pair and the mechanisms that connect them together.

It is possible to use traditional parallelization and execution techniques to improve the performance of CQ dataflows. Like parallel query plans, one can take all the operators in a CQ dataflow and horizontally partition them across the sites in a cluster. However, since continuous queries are by definition unbounded, the optimal partitioning of the state and input data stream of any particular operator is likely to change over time. If the partitioning is not adjusted, operators will perform poorly due to imbalances that develop among the sites.

Changes in the workload are one class of variations that can cause imbalances. One such change is in the input data

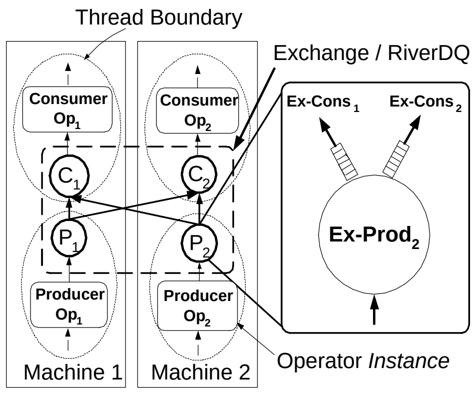


**Figure 1.** *Exchange and RiverDQ Architecture*

distribution. For example, in a network monitoring scenario, the geographic bias of client connections may vary depending on the time of day. Or in a web-log processing scenario, the popularity of pages may vary depending on the time of day. One-time sampling is inapplicable in these settings since streams are unbounded and time-varying. Another such change is in the system workload. The system may schedule additional dataflows or add operators to existing dataflows, creating both CPU and memory contention. Since a CQ operator runs for an unbounded amount of time, it cannot pin down a large amount of memory forever. So, its main memory usage also must be flexible over time. The operator must react to variations in memory availability both on a single site and across the cluster.

Variations in the underlying runtime environment are another potential cause of problems. Heterogeneities in the underlying hardware platform can lead to unexpected performance. For example, disk read bandwidth performance can vary depending on the track being read [23]. Unexpected CPU utilizations can also arise, e.g. due to the activities of daemon processes.

Finally, these problems are exacerbated in a parallel setting. As described in Section 4, a slowdown in just a single machine can cause all instances of a partitioned operator to equally under-perform. Since a dataflow pipeline is only as fast as its slowest operator, the entire pipeline may suffer.

### 2.3. Exchange

In a parallel database, an Exchange [11] operator is used to connect a stage in a pipeline where the producer's output needs to be repartitioned for the consuming operator. For example, if a relation is not partitioned on the join key, then the relation scan is connected to a hash join consumer via an Exchange. The Exchange ensures proper routing of data between the producer and consumer operators.

Although Exchange is conceptualized as a single operator, it is actually composed of two intermediate operators, Ex-Cons and Ex-Prod (see Figure 1). An Ex-Cons instance supports the traditional `getNext()` interface, and when invoked simply polls its inputs in a round-robin fashion and returns the incoming data to its consumer operator instance. An Ex-Prod instance connects each producer instance to all consumer instances via Ex-Cons. Ex-Prod encapsulates the routing logic. It calls `getNext()` on the producer instance and routes the output tuples to the appropriate consumer instance based on the tuple's content and the partitioning of the consumer operator. For example, for a hash-join, Ex-Prod would use the hash of the join key to determine the destination of the tuple. We call this type of routing *content sensitive*, since the destination is determined by the contents of the tuple. Range partitioning is another well-known content-sensitive partitioning strategy.

In an iterator architecture [12], there is typically a thread boundary at the Exchange, so the producer and consumer operator instances are scheduled independently, in separate threads (see Figure 1). The benefits of Exchange are twofold. First, the operator writer can write relational operators while being agnostic as to whether they will be used in a parallel or single-site setting. Second, since a producer instance is generating output for many consumer instances and vice-versa, placing the producer and consumer instances in different threads allows a certain degree of overlapped execution among the instances. The larger the queue between Ex-Prod and any of its Ex-Cons destinations, the more overlapped execution that is possible. If any of the outgoing queues of an Ex-Prod is full, it must block until there is available space for the next tuple.

### 2.4. RiverDQ

The distributed queue (RiverDQ) abstraction in the River [23] system addresses the load-balancing problems for a constrained set of operators: those for which the partitioning of the input stream can be *content-insensitive*, i.e. any tuple in the input stream can be sent to any instance of the consumer operator. This includes consumers such as selections, projections, and distributive aggregates. The RiverDQ architecture is the same as the Exchange except that the routing logic in Ex-Prod is modified. Instead of content-sensitive routing, a tuple is routed to a randomly chosen consumer instance weighted by the emptiness of the queue to that instance. Thus, slower consumer instances will have a larger backlog and are less likely to receive the next input. This mechanism, however, is inapplicable to operators that require *content-sensitive* routing like hash-based group-by-aggregates or joins because re-routing the input requires moving the corresponding state needed to process it.

### 2.5. Flux Contributions

Flux is a generalization of Exchange and RiverDQ that encapsulates the logic for online repartitioning of a wide range of content-sensitive operators. Flux is inserted in a dataflow stage for which the consumer operator is a potential bottleneck. The key design feature of *encapsulation*, inspired by Exchange, relieves the operator developer of the burden of implementing this logic for each query operator. The dataflow APIs to content-sensitive operators that use Flux need some modification to allow state movement, but we endeavor to make this API as lean as possible.

## 3. Experimental Methodology

In this section, we describe the experimental methodology used to illustrate the benefits of Flux mechanisms.

Throughout this paper, we use a hash-based, windowed group-by-aggregate operator as an example. It is similar to a traditional hash-based group-by-aggregate operator [3] except that it maintains a history of the most recently processed tuples for each group. In the CQ context, this operator takes a stream, splits it into multiple logical streams (one per group), and computes a statistic over the recent history of the extracted streams. Such an operator can be used, for example, in a network monitoring scenario to compute the average packet size per client over the most recent $k$ packets received. Variants of this operator exist [32] where both the statistic computed and the notion of recent history can vary.

To isolate and illustrate the effects of Flux, we restrict ourselves to a simple hash-based windowed group-by that, on every new tuple, computes a statistic over a fixed size history. In terms of resource use, the salient characteristics of this operator are similar to ones with more sophisticated definitions of history. It has bounded but non-negligible state, and the per-tuple processing time is non-trivial.

To evaluate our mechanisms, we built a simulation of this operator on a shared-nothing cluster. Each simulated node in this cluster has a 1000 MIPS CPU, with a single 80GB disk, and 1 GB of main memory. The nodes are connected by a gigabit ethernet network. We model the network connection between each pair of machines as a fixed size queue that holds two 1K packets, and each packet has a latency of $70\mu$s. The network is not the bottleneck for most of our examples. So unless otherwise specified, we assume the network has infinite bandwidth. We simulate a disk with read and write bandwidth of 20 MB/s and average seek time of 5 +/- 0.2 ms. We model the per-tuple in-memory processing time of our windowed group-by using a normal distribution with a mean of 45 +/- 3 $\mu$s. On a 1000 MIPS machine, this accounts for the time to probe the hashtable and update the history of the group; it also allows for modest additional processing for computing the next aggregate, e.g. standard deviation.

Our simulator is based on the Telegraph system [26], and the simulation of each node is a hybrid between operators in the Telegraph code base and a discrete-event simulator. Thus, we have a working implementation of the Flux APIs described in this paper. Only the hardware, aggregate function evaluation, and underlying scheduler are simulated to allow us to easily control and scale our experiments.

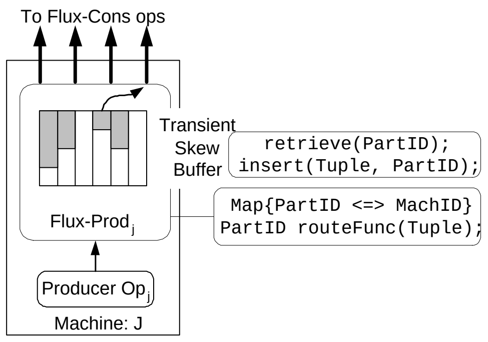


**Figure 2.** *Flux-Prod Design*

The baseline for all our experiments is a single producer-consumer stage. This stage is partitioned across a 32-node cluster and composed using a traditional Exchange. The consumer operator is our windowed group-by that maintains a history for 16K groups that are partitioned evenly across the cluster. Each machine also has a producer instance that generates an 80 byte tuple in 0.5 $\mu$s on every `getNext()` call. Thus, producer instances generate tuples as fast as the consumer operator can process them allowing us to assess the consumer's maximum sustainable performance. Unless otherwise noted, each producer instance generates tuples uniformly distributed across the 16K groups. At steady state, the aggregate throughput of the consumer operator, i.e. the total across all instances, is $6.9 \times 10^5$ tuples/sec. This setup has a cross-sectional communication bandwidth requirement of 450 mbps, well under the capacity of gigabit ethernet.

## 4. Short-Term Imbalances

In this section, we describe a buffering and reordering mechanism that allows Flux to handle short-term imbalances better than Exchange for content-sensitive consumers. Flux extends the Exchange design, and analogously, its intermediate operators are called Flux-Prod and Flux-Cons.

In a stage with an Exchange, there is typically all-to-all communication between the Ex-Prod and Ex-Cons instances. In such a case, a slowdown in a single consumer instance can cause a slowdown of the entire stage, an effect that is a result of *head-of-line blocking*. Imagine the input from each producer instance is distributed uniformly across the consumer instances. Then, all Ex-Prod instances are forced to produce at the rate of the slowest consumer instance because when a queue to that instance is filled, an Ex-Prod instance must block until that queue has available space. This effects the delivery rate to the other consumers causing each consumer instance to proceed only as fast as the slowest and thereby slowdown the entire operator.

Short-term imbalances across the cluster or transient skew can cause head-of-line blocking in an Exchange stage, resulting in potentially severe performance degradation.

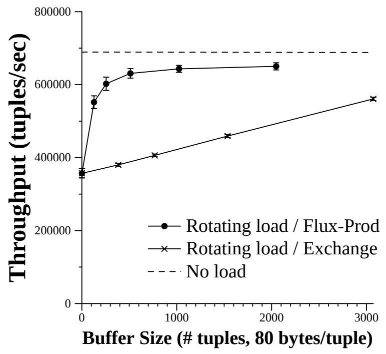


**Figure 3.** *Effect of Short-Term Imbalances*

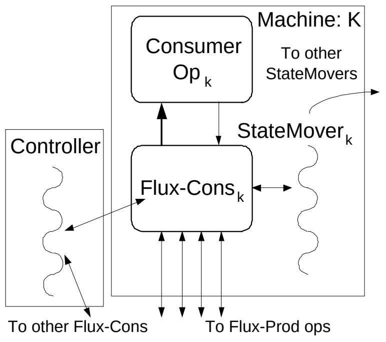


**Figure 4.** *Flux-Cons Design*

This situation can arise for various reasons. One source of transient skew is data arrival order. For example, even though the producer operator may generate tuples from a uniform distribution, it may distribute these values unevenly over time – e.g., it may generate tuples in batches that first go to machine A, then machine B, then machine C, etc. A similar effect can arise if short-term processing load cycles through the machines in a cluster in round-robin fashion. Such load may occur, for example, when a central monitor cycles through the machines in a cluster probing for statistics. In this case, every one of these perturbations impacts all producer instances in a serial fashion.

The alert reader may have noticed that when an Ex-Prod instance blocks as a result of transient skew, there may be ample queuing space in the connections to other machines. To mollify the head-of-line blocking effect, we need a more flexible data structure to take advantage of the extra space. Instead of placing the buffer space in the connections, Flux-Prod uses a buffer interposed between it and the producer operator instance. This buffer, called the *transient skew* buffer, supports the `retrieve(int PartID)` method that returns the next available tuple intended for a particular machine (see Figure 2), and a `insert(tuple T, int PartID)` method for queuing up a tuple for the destination `PartID`.

The buffer is implemented as a hashtable keyed on the destination of the tuple, and pins down a fixed amount of memory for that hashtable. It is similar to the Juggle [22] operator, except it does not spill tuples to disk. Flux-Prod drains tuples from the buffer for only the connections that can accept another tuple. Flux-Prod first calls `getNext()` on the producer instance if the buffer has space to hold another tuple, inserts the returned tuple into the buffer based on its destination, and then extracts and forwards a tuple from the buffer for an unblocked destination if one exists.

This enhancement permits reordering of the input tuples. It allows the Flux-Prod to continue to forward tuples in the buffer intended for other destinations even when the connection to a particular destination is blocked. With this design, the buffer space is flexibly allocated to the destinations on demand, as opposed to dividing the space among the destinations beforehand. If the buffer is large enough to absorb the type of transient skew described above, then the Flux-Prod instances will be affected by the largest of the perturbations, and not each one individually. In the next section, we describe how the transient skew buffer is also used to avoid blocking Flux-Prod during state movement.

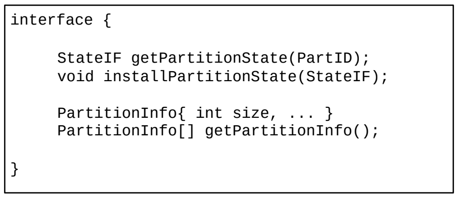
We modified our baseline configuration to use Flux-Prod with a transient skew buffer and compared it to Exchange with an equivalent amount of space in the outgoing queues. We introduced a load that goes round-robin on the machines, lasting 0.5 sec. on each machine. We simulate the load by introducing a process that consumes processing time of $100\mu$s and sleeps for $50\mu$s. On a single machine with a windowed group-by running at steady state, the additional process degrades throughput of the operator instance by about 57%. We artificially warmed up the buffer at the start, and ran the parallel simulation with the rotating load for 16 sec. (32 machines x 0.5 sec.) of simulated time. Figure 3 plots the average aggregate throughput for each run as we varied the buffer size. With no buffer, throughput of the entire stage drops to about half of the unloaded case, even though only a single machine is overloaded at any time. As buffer size is increased, Flux quickly beats Exchange and approaches the throughput of the unloaded case.

## 5. Long-Term Processing Imbalance

Long-term load imbalances eventually overload any fixed-sized buffer, so the previous technique is only suited for short-term imbalances. As mentioned before, some sources of long-term imbalance include changes in both the query mix and the underlying data distribution which can make content-sensitive operators severely under-perform. Thus, we describe a mechanism to adjust the partitioning of an operator on the fly and describe policies for deciding when and how to repartition them.

### 5.1. Flux Repartitioning Mechanism

Like the RiverDQ, we need the ability to repartition the input stream to rebalance the work for a content-sensitive operator. In this case, repartitioning the input also requires



```
interface {

     StateIF getPartitionState(PartID);
     void installPartitionState(StateIF);

     PartitionInfo{ int size, ... }
     PartitionInfo[] getPartitionInfo();

}
```

**Figure 5.** *Interface content-sensitive operators export to allow repartitioning.*

moving the accumulated operator state necessary for processing the input. For example, in our windowed group-by, if the input stream for a particular group is shifted to another machine, then the group's history needs to be relocated accordingly. To allow repartitioning of the state of a content-sensitive operator, we make a few changes to our design.

We first introduce a level of indirection. Instead of having a partition per consumer instance as in Exchange, we instantiate numerous "mini"-partitions, so that the total number of partitions is much larger than the degree of declustering of the consumer operator. Creating numerous partitions for handling skew is a well known technique [9]. These numerous, small partitions are distributed amongst the consumer operator instances at initialization time, and those instances are responsible for processing the corresponding input.

To permit reallocating these partitions for load-balancing purposes, the consumer operator must implement, in addition to the traditional iterator interface, the `getPartitionState()` and `installPartitionState()` methods for extracting and installing the state associated with a particular partition (see Figure 5). The `getPartitionState()` method extracts the partition state from the internal data structures, and marshalls the partition state into a machine-independent form. The `installPartitionState()` method unmarshalls the partition state, and installs it into the operator's internal data structures. These methods operate on a list of pages that contain the partition state. These extraction and installation methods are invoked only by Flux-Cons. We describe the purpose of the `getPartitionInfo()` method in Section 6.

Because we added a level of indirection and permit state movement, a given partition can potentially be located on any one of the sites of the content-sensitive operator. So, the routing logic in Flux-Prod needs to change. The routing function must return a partition id instead of a machine id. In addition, Flux-Prod maintains a mapping between the partitions and their sites to determine the destination of a tuple (see Figure 2). Also, the transient skew buffer is keyed on partition id instead of machine id to avoid blocking Flux-Prod during state movement as described in the next section. Flux-Prod now cycles through only the partitions for which the destination connections are unblocked.

We also modify Flux-Cons to maintain statistics about the execution and to coordinate state movement. We first describe the added mechanisms for coordinating state movement depicted in Figure 4. Each Flux-Cons instance maintains a reverse connection to all Flux-Prod instances, and a bi-directional connection to a central Controller, located on a separate dedicated machine. The Controller is responsible for collecting the runtime information from the Flux-Cons instances and issuing movement decisions for load-balancing. Also shown in Figure 4, is the StateMover thread. It is responsible for asynchronously transferring state or receiving state as instructed by Flux-Cons. Each Flux-Cons instance communicates with a local StateMover thread through a shared-memory data structure. Each StateMover thread maintains a connection to all the other StateMover threads at the other sites. Note, the StateMover thread and Controller can be shared by multiple Fluxen in a dataflow[1].

A Flux-Cons instance maintains two statistics that the Controller uses to determine if repartitioning will be effective. First, it keeps a count of the number of tuples processed per partition. Second, it estimates the amount of time the instance has spent idle. Idle time is the amount of time the Flux-Cons instance waits on the incoming connections for a tuple, and is used to determine utilization of the consumer instance. Because we need a way to estimate idle time, we require Flux-Cons and Flux-Prod to be scheduled in separate threads. In a polling-based implementation of Flux-Cons, it can estimate idle time by the time spent "sleeping" while waiting for input to arrive. The better the estimate, the more effective the load-balancing policy is in improving the aggregate performance of a content-sensitive operator.

## 5.2. State Movement Protocol

For a content-sensitive operator like our windowed group-by, the state of a partition depends upon the sequence of tuples it has processed. Thus, moving a partition requires the partition state to be consistent with respect to the sequence of tuples delivered to it. In order to ensure this consistency, our state movement protocol needs to quiesce the input stream to the partition before it is transferred. Moreover, to avoid blocking during state movement, Flux needs to buffer the input for the in-flight partitions while still continuing to process the input for stationary partitions. Below, we detail the state movement protocol and mechanisms that achieve both these criteria.

Moving a partition from one instance to another involves the following steps: quiescing the input to the partition, marshalling the state into machine independent form and removing it from the internal data structures, transferring the

[1]We pluralize Flux in the same manner as "ox" or "VAX".

state, unmarshalling the state and installing it into the receiving instance, and restarting the partition's input stream. We walk through the steps of how a partition is moved from one site to another to illustrate the details of each of these steps.

State movement is initiated by the Controller, which starts the quiescing phase. After receiving statistics from all the Flux-Cons instances, the Controller consults the repartitioning policy and generates a list of partitions to move. The moves are then sent to both the originating and destination Flux-Cons instances for each partition to be shifted. A Flux-Cons instance, once invoked by the consumer instance (through `getNext()`), first checks for messages from the Controller. When a move request is received by the destination Flux-Cons, it queues up a receive request with the StateMover thread. When a move request is received by the originating Flux-Cons, it broadcasts a *pause* request on the relevant partition to all Flux-Prod instances, and then continues to process normally. When a Flux-Prod instance receives a pause request, it marks that partition as stalled and stops draining tuples for that partition from the transient skew buffer. It then immediately sends an acknowledgement to the originating Flux-Cons instance. Assuming ordered message delivery, once the originating Flux-Cons instance receives all acknowledgements, all in-flight tuples for the candidate partition have been received and processed. At this point, the candidate partition is successfully quiesced.

Next, the originating Flux-Cons extracts the partition state from the consumer instance via the `getPartitionState()` method, and passes the result to the StateMover thread to transfer the state. Once the state has been transferred, the StateMover thread at the destination site notifies the local Flux-Cons instance. The destination Flux-Cons invokes the `installPartitionState()` method on the consumer instance, and then sends a *restart* message on the relevant partition to all Flux-Prod instances and the controller. When the *restart* message is received, the Flux-Prod instances update their map for the partitions, and resume draining tuples for the stalled partition. Note, during this protocol, state transfer across machines occurs in the background while Flux-Cons and Flux-Prod continue processing for partitions not being moved.

We made two important design decisions to facilitate state movement. First, although quiescing causes a transient delay on a single (mini-)partition, we can take advantage of the transient skew buffer to allow Flux-Prod to stall the input for in-flight partitions without blocking. The buffer holds the input for the stalled partitions while permitting Flux-Prod to continue forwarding tuples for other partitions. If we had placed the buffer space into the destination connections like Exchange, then when a tuple for a stalled partition arrived, Flux-Prod would be forced to block to avoid forwarding that tuple. Moreover, by limiting the buffer space between Flux-Prod and Flux-Cons instances, we minimize the number of in-flight tuples, and hence the time it takes to quiesce a partition.

Second, because the extraction and installation methods are called only by Flux-Cons, we are able to simplify the implementation of these routines. Since dataflow operators are typically single-threaded, no internal state is concurrently being modified once control has passed to Flux-Cons. Thus, implementing the extraction and installation methods requires only slight modification to existing operators without introducing additional synchronization overhead during normal processing.

## 5.3. Load-Balancing Policy

The goal of the Flux load-balancing policy is to repartition the consumer operator to improve aggregate performance for that stage. Since it is difficult to predict future behavior, the aim of our policy is to react to existing conditions and avoid thrashing, i.e., repeatedly making bad decisions causing the operator to make no progress. Because moving state has a significant cost, this policy tries to maximize the benefit of rebalancing while minimizing the number of partitions moved. In this section, we assume that there is ample main memory to accommodate the state of the consumer operator. Under this assumption, average latency per tuple is just the in-memory processing latency of the tuple. Thus, the metric for which to optimize is throughput.

The policy for load-balancing proceeds in rounds. Each round consists of two phases: a statistics collection phase, and a move phase. At the beginning of each round, the Controller sends a message to all Flux-Cons instances that serves as a signal to begin collecting statistics. Within that message is a duration $\delta$ indicating how long a Flux-Cons instance should accrue statistics before it returns the information to the Controller. When $\delta$ time passes, each Flux-Cons instance, $j$, returns the amount of time spent idle, $I_j$, during the collection phase, and the number of tuples processed per partition, $N_{pid}$. The Controller computes the utilization of each site (or instance), $U_j = 1 - \frac{I_j}{\delta}$, and sorts the sites by decreasing utilization. For each site, it sorts the partitions in descending order of the number of tuples processed, and calculates the total tuples processed, $T_j$. The sites with utilization above the average, $\bar{U}$, are considered (in descending order) for shedding load.

Our policy aims to equalize the utilization of each of the sites as well as possible in the fewest number of moves. Starting from the most utilized site (the donor site), we pair it with the least utilized site (the receiving site), and repeatedly pair sites at the ends as we work toward the middle. Figure 6 shows an example with four sites, sorted in descending order of utilization. The size of each partition represents the amount of processing time each partition consumed in the round. We pair site 1 with 4 and site 2 with

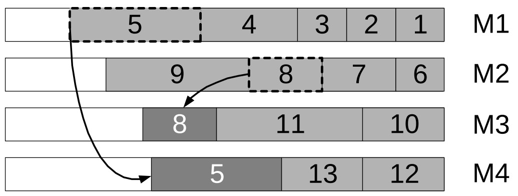


**Figure 6.** *Partition movements in a single round*

3. For each pair, we consider moving a partition from the donor to the receiver. We first apply a threshold test. If the donor's utilization, $U_d$ is less than the average, $\bar{U}$, or if the utilization ratio (i.e. imbalance) of the two sites is less than some threshold factor $th_{imb}$ or if the receiver's utilization, $U_r$, is above a threshold, $th_{util}$, then we do not consider those sites for any state movement. Otherwise, we walk down the sorted list of partitions and schedule a move for the first partition in the donor's list that will reduce the utilization imbalance between the donor and receiver.

To estimate the post-repartitioning utilization of each site, we assume that the number of tuples processed by each partition and the processing rate of each consumer instance (or site) remains fixed across rounds. Given a partition to be moved that processed $N_p$ tuples, the donor's new utilization is estimated as $U'_d = U_d\left(1 - \frac{N_p}{T_d}\right)$. Likewise, the receiver's new utilization is estimated as $U'_r = U_r\left(1 + \frac{N_p}{T_r}\right)$. The second term in both equations is an estimate of the change in utilization given the current processing rate of the consumer instances. If the receiver becomes overutilized, $U'_r > 1$, then we do not schedule the move.

Figure 6 shows an example in which the partitions are sorted by the number of tuples processed. Partition 5 is moved to site 4, and partition 8 is moved to site 3 because moving 9 would increase the imbalance between sites 2 and 3. Only a single partition is moved for every pair during a round. We continue down our list of machine pairs until all pairs have been considered or some pair fails the threshold test. During any round at most $d/2$ partitions are moved where $d$ is the degree of declustering for the operator, and each site moves or receives at most one partition.

Once the moves have been determined, the Controller sends out the move orders to the appropriate Flux-Cons instances and starts a timer. Once all the moves have completed the round is finished, and we use the timer to record the duration of the move phase. The duration of the next collection phase $\delta$ is set to the length of the previous move phase. This heuristic ensures that even if we make poor movement decisions, at most half the time is spent stalling and repartitioning. In effect, this heuristic dampens the load-balancing policy and prevents it from overreacting. In case no partitions were moved, we set $\delta$ to half its previous value. Our collection phase length is subject to a minimum, $\delta_{min}$. In the next section, we show this minimum collection phase duration allows the system to be configured to avoid excessive movements when partition sizes are small.

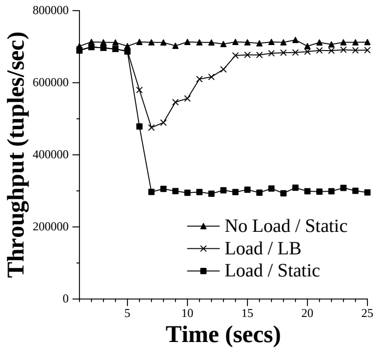


**Figure 7.** *Balancing Processing load*

### 5.4. Experiments w/ Processing Load

To show the effectiveness of our load-balancing policy, we ran a number of simulations of the windowed group-by operator using Flux. We instantiate 64 partitions per machine, with each partition holding a history of 10,000 tuples, or 800KB. In this example, even though the entire state of the windowed groupby can fit onto 2 machines, we utilize the cluster for improved throughput. Each producer instance generates tuples uniformly distributed across 16K groups. Unless otherwise specified, for all subsequent experiments, the Flux-Prod transient skew buffer is set to 160KB (2048 tuples). Also, we charge 0.2 $\mu$s per tuple for quiescing and installing partitions. To approximate the bandwidth limitations of a gigabit ethernet network during the move phase, we assume that the maximum available cross-sectional bandwidth is 500mbps for partition movement, and maximum point-to-point bandwidth is 250mbps. For the load-balancing policy, we used the following parameter values $\delta_{min} = 10ms, th_{imb} = 1.2, th_{util} = 0.9$.

Our first experiment examines how quickly the load-balancing policy reacts to a load perturbation by introducing an external load on a single machine. As before, an extra process that computes for 100$\mu$s and sleeps for 50$\mu$s is used to exert the load. The top line in Figure 7 shows performance the with no machines perturbed. The bottom curve shows the performance with load-balancing turned off and load introduced at $t = 1$ sec. We report the aggregate throughput computed over one second intervals. The effect of additional load is felt only after $t = 5$ sec.: head-of-line blocking ensues when the transient skew buffer fills, affording us about 4 sec. of slack. Likewise, our load-balancing policy feels the effect of the imbalance at the same time as shown in the middle curve. However, it begins to move partitions to react to this perturbation. It reaches steady state after offloading 32 partitions at $t = 14$ sec., 9 sec. from when it began rebalancing. The minimum time needed to

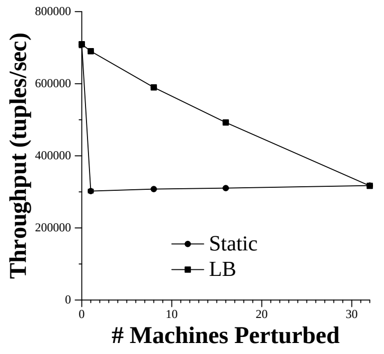


**Figure 8.** *Graceful Degradation*

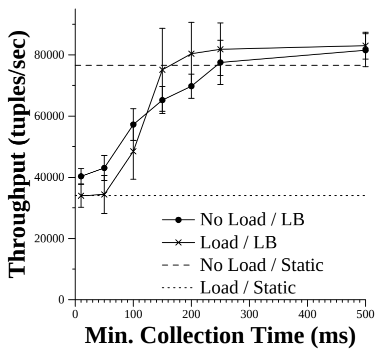


**Figure 9.** *Varying Collection Time, $\delta_{min}$*

transfer these partitions is 2 sec. The dilation in reaction time is a result of two factors. One is the time to process and drain in-flight tuples before partition movement occurs, and another is the collection phase which waits as long as the previous move phase.

Our second experiment displays how aggregate performance degrades with increasing external load. We perturbed a random subset of the machines in the cluster with the same external load and report the average *steady state* aggregate throughput of the consumer operator, i.e., after the load is felt in the static case, and after rebalancing has completed in the load-balancing case. The bottom line in Figure 8 shows the performance without the load-balancing policy. As expected, a result of the head-of-line blocking phenomenon described before, the throughput of the entire stage drops once a single machine is perturbed and remains there. On the other hand, the load-balancing policy manages to rebalance the partitions thus exhibiting a smooth, linear degradation profile.

Finally, we present experiments in which the policy overreacts and exhibits suboptimal performance, and we show how increasing the minimum collection time parameter $\delta_{min}$ can prevent it from overreacting. We modified the producer instances to generate tuples according to an 80/20 distribution across 16K groups. Using this workload and no external load, we initially ran two types of experiments, with and without the load-balancing policy. For all experiments, we ran the simulation for 60 sec. of simulation time, computed the aggregate throughput in one second intervals, and report the average of the 60 values. Figure 9 shows at small values of $\delta_{min}$ the load-balancing policy achieves half the throughput of the static case. In this skewed workload, there are a few partitions that receive most of the work and are the bottleneck. Since the partitions are small, movement time is short, causing the collection phase to be short. During collection, the controller only receives an accurate sample of processing load on machines with popular partitions; hence, it keeps shifting popular partitions. As $\delta_{min}$ is increased, it dampens the frequency of movements, and at $\delta_{min} = 250$ ms, the load-balancing policy outperforms the unloaded, static case. We reran the same experiments with the same external load introduced on 8 machines, ensuring that the site initially with the most popular partition was also overloaded. As expected, in the static case, the throughput drops and the load-balancing policy is still ineffective at small $\delta_{min}$. As $\delta_{min}$ is increased, its performance quickly improves, and it outperforms the static unloaded case at $\delta_{min} = 150$ ms. Surprisingly, the policy works better (i.e., it requires a smaller $\delta_{min}$) in the loaded case than in the unloaded case. In the loaded case, the policy moves popular partitions onto unloaded sites quickly because the imbalance between sites is larger, and the load imbalance prevents the partitions from being moved back. Of course, as we noted earlier, the trade-off for increasing the collection phase duration is that it takes longer to rebalance. Because the policy is sensitive to its parameters, autotuning of these values is an interesting and necessary direction for future work.

## 6. Memory-Constrained Environment

In the previous sections, we only considered repartitioning in conditions with ample aggregate memory, but for content-sensitive operators often memory can be the critical resource rather than processing capacity. For example, if our windowed group-by is required to maintain a much larger history or the number of groups increases dramatically, then it could run short on physical memory for its internal state. Or, the CQ runtime system could instantiate another operator on the same machines, forcing existing operators to release some of their memory. When physical memory is exhausted on some site, content-sensitive operators like our windowed group-by have three choices: shed state to disk, move state to another site, or decrease the history size maintained. In certain critical applications like intrusion detection or monitoring stock quotes, the last option may not be acceptable. We focus on such applications in which content-sensitive operators must adapt their memory usage by repartitioning to employ both local disk and available memory across the cluster.

Flux uses a dual-destination repartitioning mechanism for memory-constrained environments. In this section, we first describe changes to Flux-Cons that facilitate spilling to local disk and then describe changes to the Controller that provide a global repartitioning policy to efficiently use aggregate main memory. Flux-Cons leverages the state movement APIs to provide a local, per-site mechanism for spilling partitions to disk. Since streaming operators have unbounded input, Flux uses a local round-robin style policy that continually rotates through on-disk partitions to avoid accruing input tuples indefinitely. When an operator starts spilling, both throughput and average latency per tuple are degraded. Throughput is diminished because I/O is performed, and latency rises because input tuples for on-disk partitions must be spooled to disk and processed later. Thus, imbalances in memory usage across a cluster can lead to suboptimal aggregate performance in both respects. The global repartitioning policy balances memory use across the cluster to avoid or postpone local disk use and thereby improve both throughput and average latency.

### 6.1. Local Mechanism: Spilling State and Input Data

In this section, we describe how Flux-Cons leverages the existing interface for extracting and installing partitions to provide a reusable mechanism for spilling partitions and corresponding input tuples to disk in memory-constrained environments. Since in a CQ environment the input stream is unbounded, input tuples for spilled partitions can accrue indefinitely. This precludes the two-phase spilling schemes used in traditional out-of-core hashing and sorting. Instead, we describe a local policy that Flux-Cons implements to continually rotate through on-disk partitions, processing the spilled tuples.

The model we assume for memory-adaptive operators is that they have a free pool of pages that can be used for internal state. When the runtime system has additional memory to donate to the operator, it invokes a `donatePages()` method to pass (references to) pages that remain pinned in memory. When the runtime system needs physical memory, it posts an asynchronous request to the operator to relinquish a certain amount of pages through a `postRelinquish()` method. Upon a relinquish request, such an operator returns free pages if available via a `releasePage()` method. If not, it immediately forces some portion of its internal state to disk, releases the pages, and continues processing the incoming input.

We modify the Flux-Cons operator to implement the `donatePages()` and `postRelinquish()` methods, and maintain a per-operator memory pool (see Figure 10). The memory pool implements the `freePage()` and `newPage()` methods so the consumer operator can shuffle pages to and from the pool, respectively. The consumer operator must implement a `getPartitionInfo()` method, invoked by Flux-Cons, that returns the sizes of all installed partitions (see Figure 5). The repartitioning policy uses this partition size information to coordinate partition movement. The operator developer is relieved of implementing the `donatePages()` and `postRelinquish()` methods which may be invoked asynchronously by the runtime.

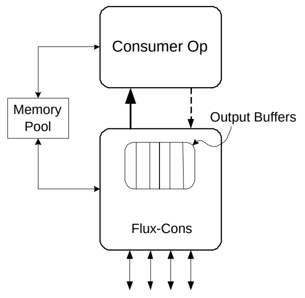


**Figure 10.** *Memory-Adaptive Flux-Cons*

To service these requests, Flux-Cons monitors the free pool size. If the memory pool is low on pages (drops below a threshold), or a relinquish request arrives and additional pages are needed, it uses the `getPartitionState()` method to remove partitions, writes them to disk, and releases the pages to the runtime system. Moreover, when ample space is available, on-disk partitions are retrieved and installed via the `installPartitionState()` method. Flux-Cons also maintains a set of output buffers for spilling input data to contiguous per-partition runs on disk. Similar to the hybrid hash join[12], it performs random I/Os while spilling runs, and sequential I/Os when reading them back in. Another advantage of these buffers is that they speed up the state movement protocol. Instead of requiring the consumer instance to process every in-flight tuple that precedes the pause acknowledgements, Flux-Cons can immediately drain the connections and hold the in-flight tuples in the output buffers for later processing.

As a default, Flux provides a round-robin style local policy for processing pending input for on-disk partitions. This policy proceeds as follows. There are two FIFO queues maintaining a list of in-memory (active) and on-disk (inactive) partitions. If the most recently activated partition has been active for longer than time $t_{gap}$, then the policy initiates a swap of partitions from memory to disk. The first partition in the in-memory queue is extracted via `getPartitionState()`, written to disk, and added to bottom of the on-disk queue. Then, the first on-disk partition is read, added to the bottom of the in-memory queue, and then installed or activated via the `installPartitionState()` method. Once an on-disk partition is activated, all of its pending input is processed. When all pending input is processed, Flux-Cons continues normal processing until $t_{gap}$ time has passed again since the most recently read partition was activated. Then, it repeats.

With this policy, the value of $t_{gap}$ allows us to trade off between throughput and average latency per tuple given a fixed amount of memory. The larger the value of $t_{gap}$, the smaller fraction of time Flux-Cons spends doing I/O, leading to higher throughput. However, a larger value also means the time between deactivation and activation is longer for any given partition, so more tuples will be spilled to disk causing average latency per tuple to increase. A CQ system can use this parameter to adjust to quality-of-service requirements specified by a user. As a rule of thumb, in our experiments, we set $t_{gap}$ equal to the time taken to read in the previously activated partition.

### 6.2. Global Memory-Constrained Repartitioning

Like the case with load imbalances, even a single instance that exhausts memory and spills to disk can hamper the performance of the entire consumer operator. To detect such imbalances under heavy memory pressures, Flux needs additional interfaces to the consumer operator to determine its memory usage characteristics. The idle time metric fails to accurately reflect imbalances when available memory is exhausted because the local Flux-Cons mechanism is busy performing I/O, so it rarely exhibits idle time.

Thus, Flux-Cons leverages interfaces to the memory pool and the `getPartitionInfo()` method for memory-based repartitioning across the cluster. By using these new interfaces, the Flux-Cons operator returns to the Controller information about memory availability and partition sizes. Since it also handles spilling, Flux-Cons also indicates whether partitions are memory resident, and the size of their unprocessed input. The Controller implements a global repartitioning policy that attempts to balance memory usage amongst machines and balance disk usage when aggregate memory is exhausted.

To deal with the memory-constrained case, we modify the policy in Section 5.3 to use the memory usage information directly for repartitioning. The policy continues to proceed in rounds with a collection phase and move phase, but instead of considering idle time, the Controller determines the excess memory capacity, $total\_state\_size - mem\_avail$, of each site (or consumer instance). Sites are sorted in increasing order of excess memory capacity. Note, excess capacity is negative when the free page list is exhausted and partitions are on disk. For each site, the partitions are sorted in decreasing order of size, with in-memory partitions listed or prioritized before on-disk partitions, because the latter require I/O and processing spilled input before movement. Like the load-balancing policy, we select partitions to move by pairing sites with less excess capacity to sites with more excess capacity. For each pair, we traverse down the list of partitions and move the first that reduces the imbalance in excess capacity between the two. In the threshold test, we remove the utilization restrictions for the receiver, because although memory may be exhausted, movement would balance the number of on-disk partitions.

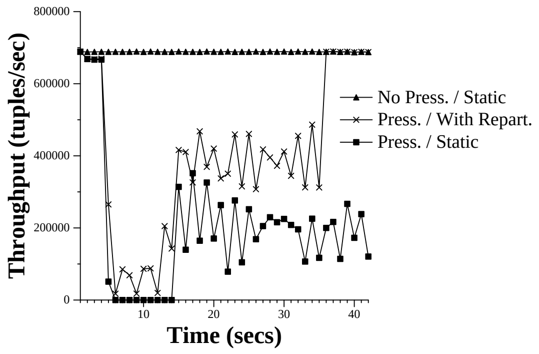


**Figure 11.** *Throughput during Memory Balancing*

### 6.3. Experiments w/ Memory Pressure

To illustrate the efficacy of the memory-constrained repartitioning policy, we ran a number of simulation experiments and analyzed their behavior with respect to throughput and average latency per tuple. We define latency the time a tuple enters the Flux-Cons operator to the time it is processed by the consumer operator. We continue with our simulation of the windowed group-by operator using the both the local and global mechanism and policy. The size of each partition is increased to 6MB and we create 128 partitions per machine. Thus, at each site, 768MB out of 1GB is initially occupied. We simulate synchronous disk I/O for spilling to disk during which a Flux-Cons instance is blocked and not processing incoming requests.

In the first experiment, shown in Figures 11 and 12, we examine how quickly the repartitioning policy reacts to memory pressure. We ran the simulation and introduced memory pressure on a single machine by reducing the available memory from 1GB to 512MB. We report the aggregate throughput and average latency per tuple computed over one second intervals. At $t = 1$ sec., available memory is reduced, and Flux begins to extract and spill partitions to disk. It needs to extract and write 256MB to disk, which takes approximately 14 sec. to complete. During that time, the Flux-Cons operator is completely stalled, causing the producers to block and grinding throughput to a halt. In the static case, at $t = 15$ sec., the overloaded machine begins to process incoming tuples again at a much reduced rate, because it is cycling through the on-disk partitions (see Figure 11). On the other hand, when the repartitioning policy in effect, while some partitions are being transferred to disk, others are concurrently transferred to other machines. Thus, we observe some minute throughput during the stalled phase. At $t = 15$ sec., the overloaded machine starts processing incoming tuples again and the repartitioning policy continues to offload on-disk partitions to other machines. During this time, average latencies are high (see Figure 12) because activated partitions are processing their previously spilled input. Once all the partitions are in memory at $t = 37$ sec., the throughput of the stage reaches the unloaded case, and

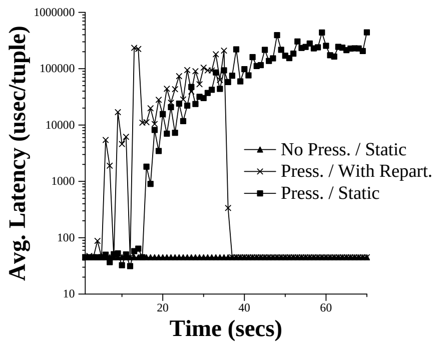


**Figure 12.** *Avg. Latency during Memory Balancing*

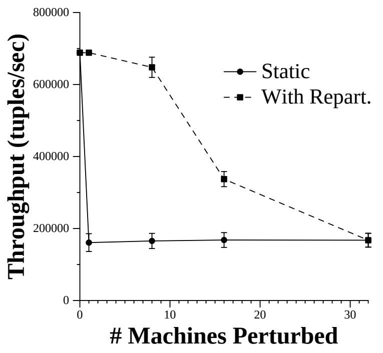


**Figure 13.** *Throughput Degradation*

average latency returns to the in-memory latency.

The repartitioning policy takes 36 sec. to reach steady state while it should only take 7.9 sec. to transfer the partitions and rebalance. This dilation in reaction time occurs because the local mechanism in Flux-Cons spills partitions to disk during which time it cannot dispatch move requests or process collection requests. This elongates both the move and collection phase in a round resulting in an increase in reaction time.

In a second set of experiments we varied the number of machines perturbed with the same external memory pressure to show how the system degrades. We report average aggregate performance in *steady state*, i.e., after the effect of the load is felt in the static case, and after rebalancing has completed in the adaptive repartitioning case. Figure 13 shows the throughput degradation as we vary the number of machines perturbed. Again, in the static case, as soon as a single machine is perturbed, the performance of the entire stage degrades. The throughput degrades because random I/Os are performed and partitions are swapped back and forth from disk. The average latency per tuple increases by several orders of magnitude (see Figure 14). This precipitous increase occurs because, at steady state on a perturbed machine, 42 out of 128 partitions are on disk. Because the partitions are so large it takes on average $0.9 sec$ for the processing of a swapped-in partition to finish and the next one begin. The average latency per tuple is proportional to the

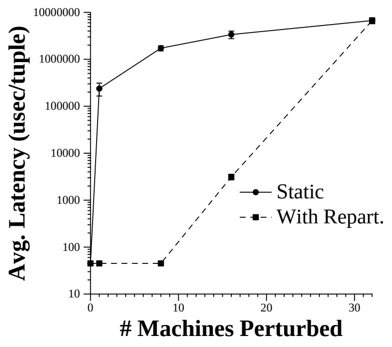


**Figure 14.** *Avg. Latency Degradation*

time it takes to cycle through all partitions on disk. Hence, we observe orders of magnitude increase in latency. With the repartitioning policy in effect, the system is able to utilize the aggregate main memory of the entire cluster. When 16 machines are perturbed, *aggregate memory* is exhausted, and each overloaded machine has one partition on disk. At this point, the throughput of the operator drops steeply and the average latency increases exponentially.

### 6.4. Hybrid Repartitioning Policy

In this section, we describe a simple hybrid that combines the previous policies, and we report experiments that show the hybrid's benefits under a mixture of external processing load and memory pressure.

The hybrid combines our load-balancing (see Section 5.4) and memory-constrained repartitioning (memory-based) policy (see Section 6.3) as follows. If any operator instance has partitions spilled to disk, the hybrid policy resorts to the memory-based policy to avoid high average latencies. If partitions for all instances are resident in memory, it resorts to the load-balancing policy to improve throughput, subject to the constraint that a partition is moved only if sufficient memory exists at the destination site.

Our first set of experiments exert a medium level of external processing and memory pressures and show that the hybrid and load-balancing policies outperform the memory-based policy. We introduce both processing and memory pressures on 8 randomly chosen machines running the baseline experiment. The windowed group-by starts out with 128 partitions per machine, and each partition is 6MB, for a total of 768MB per machine. We simulate external load with a process that processes for 100 $\mu$s, sleeps for 50 $\mu$s, and pins down 512MB of main memory. In these experiments, we introduced load at the start and ran the experiments for 100 sec of simulated time. Figure 15(a) shows the average *steady state* aggregate throughput of the consumer operator for each of the policies once partition movement had subsided. The memory-based policy performs worse than the other two because once all partitions are in mem-

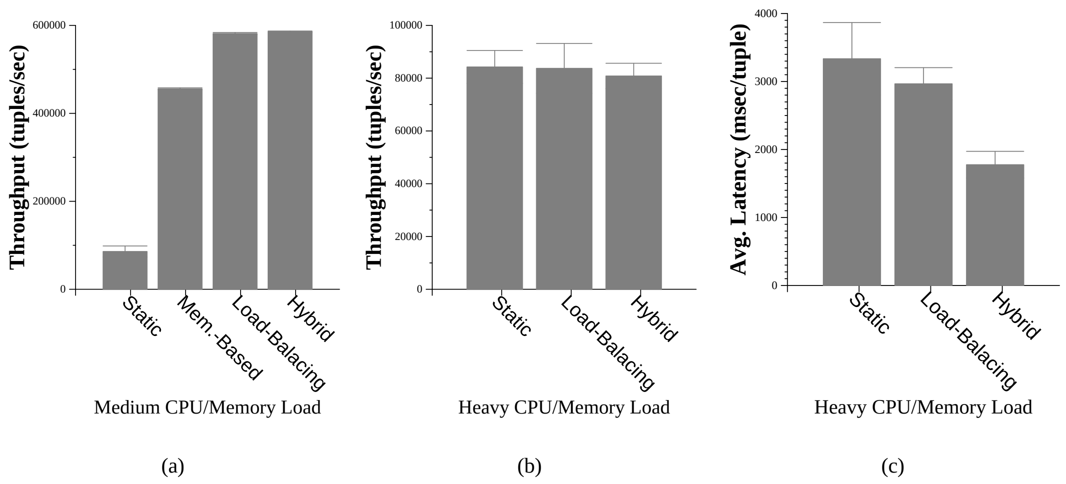


**Figure 15.** *Comparison of policies under different external loads.*

ory and evenly distributed, it stops rebalancing even though a load imbalance still exists. The hybrid and load-balancing policy continue to detect processing load on the overloaded machines and offload partitions. In addition, the hybrid policy reaches steady state (in 55 sec) faster than the load-balancing policy (63 sec) because the hybrid can directly leverage memory usage and partition size statistics early on.

In our second set of experiments, we introduce heavy external processing and memory pressures and show that the hybrid policy outperforms the load-balancing policy. Using the same initial configuration, we introduce external processing and memory load on all the machines. But, in this case, the additional process consumes 512MB on half the sites, and 262MB on the other half, just enough to cause a single partition to be spilled to disk on those machines. Note, in this setup, the hybrid policy always defaults to the memory-based policy since aggregate memory cannot accommodate the state of the entire windowed group-by. We introduced load at the start and ran the experiments for 100 sec of simulated time. Figures 15(b) and 15(c) show the average aggregate throughput and average latency per tuple for the consumer operator over the last 20 seconds of the simulation. During this period, the load-balancing policy is still periodically moving partitions, but the hybrid policy is in steady state. We see that the throughput of the operator is unaffected by either of the policies because the processing load is evenly distributed, and I/Os for rotating through the partitions are the main bottleneck. However, the hybrid policy manages to reduce the average latency by a factor of two by evenly spreading spilled partitions. The load-balancing policy is ineffective because the difference in idle time between machines is not sufficient to trigger enough partition movement to rebalance.

In summary, instead of relying on a single statistic (either CPU utilization or memory usage) to infer the detrimental effects of a combination of load and memory usage imbalances, the hybrid uses both directly to repartition the consumer. Hence, it is able to match or outperform the load-balancing and memory-constrained repartitioning policies under a variety of external loads.

## 7. Related Work

The related work which inspired Flux falls into two main categories. First, we highlight recent advancements in continuous query and stream processing systems. Next, closely related to the topic of repartitioning across a cluster, we outline past work in parallel query processing and adaptive out-of-core operators.

### 7.1. Continuous Query and Data Stream Systems

Previous CQ systems have dealt with scalability requirements by taking advantage of the commonality amongst queries. NiagraCQ [7] and XFilter [2] are examples of CQ systems over streaming XML documents. CACQ [16] is an adaptive system for relational-style continuous queries over streaming sensor data. PSoup [6] extends CACQ to accommodate disconnected operation. A key feature of these systems is to treat queries like data and index them to take advantage of their commonality. The latter two systems employ an Eddy [24] to dynamically adapt a query plan to changing workload characteristics.

STREAM [18] is a data stream processing system which focuses on computing approximate results and minimizing the memory footprint of queries over data streams. The Aurora [4] system proposes to use user specified quality-of-service profiles sacrificing result quality in the absence of sufficient resources for scalability. In contrast, we believe that inexpensive shared-nothing parallelism is a more compelling approach to scalability than specialized techniques that sacrifice result quality, though a combination of these is certainly natural. For example, internet based services use shared nothing parallelism as the workhorse for handling the common case workload and rely on admission control to degrade smoothly during unanticipated, peak loads.

While none of these systems have explicitly considered parallelism, integrating Flux into systems that are composed of fixed dataflows like NiagaraCQ, XFilter, STREAM, and Aurora, should be straightforward. The integration of Flux's enhanced memory management with the memory-sensitive designs of Aurora and STREAM seem promising. On the other hand, Eddies take a dramatic departure from traditional dataflow processing techniques, and integrating their mechanisms with Flux is a challenging future direction.

### 7.2. Query Processing

We refer the reader to Graefe's survey [12] for an overview of database query processing techniques and highlight the most relevant work on parallel query processing. Early work concentrated on parallelizing individual, traditional content-sensitive operators like hybrid-hash join [25] and sort (e.g., [10, 20, 1]). The abstractions which inspired Flux, Exchange [11] and RiverDQ [23], were proposed to compose such operators into a dataflow. Shatdal and Naughton [27] describe how to leverage shared-virtual memory across a shared-nothing cluster to implement hybrid hash join. DeWitt et. al. present practical techniques for handling data skew for a hash join and external sort [9, 10]. These techniques rely on sampling a static data set, which is infeasible in the streaming scenario. Mehta and DeWitt [17] and Rahm and Marek [21] describe how to account for current CPU utilization, memory usage, and I/O load to perform site selection and determine degree of declustering for hash joins. None of these previous schemes consider repartitioning the join operator during execution. Wolf et. al. [31] and Lu and Tan [15] describe techniques to repartition a traditional hash join at one point in time: between the build and probe phases of the join. These techniques are specific to an implementation of the hash join operator, and do not react to external load and memory pressures. Moreover, the last two papers do do not address the issue of balancing load while the build or probe phase is in flight, and hence are inapplicable to the unbounded pipelines of CQ systems.

The ConQuest [19] system takes an approach complementary to ours. The system combines a customizable rule-based system that invokes a dynamic optimizer for reconfiguring parallel database query plans at runtime. This work concentrates on mechanisms for reconfiguring the entire query plan (inter-operator optimizations) instead of repartitioning a single stage (intra-operator optimizations).

Finally, the inspiration for building spilling mechanisms into Flux arose from the literature on memory-adaptive operators like the partially preemptive hash join [14] and XJoin [29]. Pang et. al. [14] describe a join operator that adapts to fluctuations in available memory on a single site. The XJoin [29] is a symmetric hash-join variant that has a similar flavor; it adapts memory usage to variations in arrival of the input stream. None of these schemes consider adapting to memory usage across a shared-nothing cluster.

## 8. Conclusion and Future Work

Pipelined dataflows in CQ systems are bound to experience fluctuations in resource availability while executing. To perform efficiently when parallelized across a cluster, their constituent dataflow operators must adapt on the fly to imbalances that arise. In this paper, we propose a reusable mechanism, Flux, that encapsulates adaptive repartitioning for a wide range of content-sensitive operators. Flux extends the Exchange operator to include mechanisms that alleviate short-term and long-term imbalances across a cluster. Our experiments show that both CPU and memory load imbalances can cause severe performance degradation and we propose policies to accommodate both. When Flux employs these policies, it outperforms the statically partitioned case by several factors in throughput and orders of magnitude in average latency. A simple hybrid of these policies allows Flux to efficiently rebalance under various mixtures of external processing loads and memory pressures.

Given the Flux mechanism, there are a few fruitful directions for future research. As shown in Section 5.4, our policies are sensitive to their parameter values, e.g. the minimum collection phase length. A scheme for auto-tuning these parameters is necessary to avoid placing the burden on an administrator or operator designer. Investigating how these per-operator policies effect global performance in a dataflow with multiple Fluxen is also an interesting direction.

The repartitioning policies described in this paper are aimed at providing improved aggregate throughput and minimizing average latency. Developing a repartitioning policy that targets more relaxed user-specified quality-of-service (QoS) requirements should allow the system to degrade more smoothly under unanticipated peak loads. For example, in an intrusion detection scenario, analyzing the packets from specific clients may be more important than analyzing others, which may be dropped when resources are scarce. QoS denotes a minimum level of service that the system strives to maintain and can be specified in a number of ways, e.g. according to latency tolerance, throughput requirements, or value-based requirements [4].

Another interesting direction is to integrate Flux with an Eddy [24]. An Eddy is an adaptive mechanism that provides inter-operator adaptivity. It dynamically reorders the order of operators applied in a pipelined query plan on a per-tuple basis. Because Flux provides intra-operator adaptivity, combining it with an Eddy would result in a fully adaptive parallel dataflow solution. There are two natural approaches for this integration: declustering individual operators in an Eddy with Flux, or embedding the Flux mechanisms within the Eddy itself. The feasibility, applicability, and implications of these alternatives is still unclear.

Since CQ systems are expected to provide 24x7 continuous operation, faults in the underlying software and hardware platform are bound to cause existing queries to fail. For certain applications like phone call record management, simply restarting the query is not sufficient – it may be unacceptable for the system to miss charging for an important call. Depending on the availability requirements of a query, redundancy needs to be employed to avoid data loss and provide quick fail-over. Extensions to the version of Flux described here should support replication of flows, state, and controller responsibilities. We intend to integrate Flux load-balancing mechanisms with fault-tolerance mechanisms to provide controlled degradation for CQ dataflows in the face of machine failures.

## Acknowledgements

We thank Amol Deshpande and Remzi Arpaci-Dusseau for invaluable feedback that helped guide this research. Without Remzi's optimism and support this work may never have been completed. We thank Megan Thomas, Sam Madden, Owen Cooper, and the anonymous reviewers who reviewed this paper and gave useful suggestions for improving its presentation.

## References

[1] A. C. Arpaci-Dusseau et al. High-Performance Sorting on Networks of Workstations. In *SIGMOD*, 1997.

[2] M. Atinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB*, 2000.

[3] K. Bratbergsengen. Hashing Methods and Relational Algebra Operations. In *VLDB*, 1984.

[4] D. Carnery et al. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*, 2002.

[5] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. *CIDR*, 2003.

[6] S. Chandrasekaran and M. J. Franklin. Streaming Queries over Streaming Data. In *VLDB*, 2002.

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

[8] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, June 1992.

[9] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *VLDB*, 1992.

[10] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *PDIS*, 1991.

[11] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*, 1990.

[12] G. Graefe. Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys*, 2002.

[13] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. In *PDIS*, 1991.

[14] HweeHwa Pang and Michael J. Carey and Miron Livny. Partially Preemptive Hash Joins. *SIGMOD*, pages 59–68, 1993.

[15] H. Lu and K.-L. Tan. Dynamic and Load-balanced Task-Oriented Datbase Query Processing in Parallel Systems. In *EDBT*, 1992.

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

[17] M. Mehta and D. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. In *VLDB*, 1995.

[18] R. Motwani et al. Query Processing, Approximation, and Resource Management in a Data Stream Management System. *CIDR*, 2003.

[19] K. Ng, Z. Wang, R. Muntz, and S. Nittel. Dynamic Query Re-optimization. In *SSDBM*, 1999.

[20] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet. AlphaSort: A RISC Machine Sort. In *SIGMOD*, 1994.

[21] E. Rahm and R. Marek. Dynamic Multi-Resource Load-Balancing in Parallel Database Systems. In *VLDB*, 1995.

[22] V. Raman, B. Raman, and J. M. Hellerstein. Online Dynamic Reordering for Interactive Data Processing. In *VLDB*, 1999.

[23] Remzi Arpaci-Dusseau et. al. Cluster I/O with River: Making the Fast Case Common. *IOPADS*, May 1999.

[24] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. *SIGMOD*, 2000.

[25] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *SIGMOD*, 1989.

[26] M. Shah, S. Madden, M. Franklin, and J. Hellerstein. Java Support for Data Intensive Systems: Experiences Building the Telegraph Dataflow System. *SIGMOD Record*, Dec. 2001.

[27] A. Shatdal and J. F. Naughton. Using Shared Virtual Memory for Parallel Join Processing. In *SIGMOD*, 1993.

[28] M. Stonebraker. The Case for Shared Nothing. *IEEE Database Engineering*, Mar. 1986.

[29] T. Urhan and M. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, pages 27–33, 2000 2000.

[30] M. Wagner. Google Bets the Farm On Linux. *http://www.internetwk.com/lead/lead060100.htm*, 2000.

[31] J. Wolf, D. Dias, P. Yu, and J. Turek. An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew. In *ICDE*, 1991.

[32] Y. Zhu and D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *VLDB*, 2002.