Highly Available, Fault-Tolerant, Parallel Dataflows

Mehul A. Shah*
U.C. Berkeley
mashah@cs.berkeley.edu

Joseph M. Hellerstein
U.C. Berkeley
Intel Research, Berkeley
jmh@cs.berkeley.edu

Eric Brewer
U.C. Berkeley
brewer@cs.berkeley.edu

ABSTRACT

We present a technique that masks failures in a cluster to provide high availability and fault-tolerance for long-running, parallelized dataflows. We can use these dataflows to implement a variety of continuous query (CQ) applications that require high-throughput, 24x7 operation. Examples include network monitoring, phone call processing, click-stream processing, and online financial analysis. Our main contribution is a scheme that carefully integrates traditional query processing techniques for partitioned parallelism with the process-pairs approach for high availability. This delicate integration allows us to tolerate failures of portions of a parallel dataflow without sacrificing result quality. Upon failure, our technique provides quick fail-over, and automatically recovers the lost pieces on the fly. This piecemeal recovery provides minimal disruption to the ongoing dataflow computation and improved reliability as compared to the straight-forward application of the process-pairs technique on a per dataflow basis. Thus, our technique provides the high availability necessary for critical CQ applications. Our techniques are encapsulated in a reusable dataflow operator called Flux, an extension of the Exchange that is used to compose parallel dataflows. Encapsulating the fault-tolerance logic into Flux minimizes modifications to existing operator code and relieves the burden on the operator writer of repeatedly implementing and verifying this critical logic. We present experiments illustrating these features with an implementation of Flux in the TelegraphCQ code base [8].

1. INTRODUCTION

There are a number of continuous query (CQ) or stream processing applications that require high-throughput, 24x7 operation. One important class of these applications includes critical, online monitoring tasks. For example, to detect attacks on hosts or websites, intrusion detection systems reconstruct flows from network packets and inspect the flows' contents [20, 27]. Another example is processing a stream of call detail records for telecommunication network management [3, 17]. Phone billing systems perform various data management operations using call records to charge or even route phone calls. Websites may analyze click streams in real-time for user targeted marketing or site-use violations [32]. Other applications include financial quote analysis for real-time arbitrage opportunities, monitoring manufacturing processes, and instant messaging infrastructure. Recent research suggests that we can implement these applications using a more general CQ infrastructure that produces continuously processing dataflows [7, 8, 9, 26]. A CQ dataflow is a DAG in which vertices represent operators and edges denote the direction in which data is passed.

For such critical, long-running dataflow applications, scalability, high availability, and fault-tolerance are the primary concerns. A viable approach for scaling these dataflows is to parallelize them across a shared-nothing cluster of workstations. Clusters are a cost-effective, highly-scalable platform [2, 10] with the potential to yield fast response times and permit processing of high-throughput input streams. However, since these dataflows run for an indefinite period, they are bound to experience machine faults on this platform, either due to unexpected failures or planned reboots. A single failure can defeat the purpose of the dataflow application altogether. These applications cannot tolerate losing accumulated operator state, dropping incoming or in-flight data, and long interruptions in result delivery. In an intrusion detection scenario, for example, any one of these problems could lead to hosts being compromised. For financial applications, even short interruptions can lead to missed opportunities and quantifiable revenue loss. Hence, in addition to scalability, such applications need a fault-tolerance and recovery mechanism that is fast, automatic, and non-disruptive.

To address this problem, we present a mechanism called Flux that masks machine failures to provide fault-tolerance and high availability for dataflows parallelized across a cluster. Flux has the following salient features.

Flux interleaves traditional query processing techniques for partitioned parallelism with the process-pairs [15] approach of replicated computation. Typically, a database query plan is parallelized by partitioning its constituent dataflow operators and their state across the machines in a cluster, a technique known as partitioned parallelism [25]. The main contribution of Flux is a technique for correctly coordinating replicas of individual operator partitions within a larger parallel dataflow. We address the challenges of avoiding long stalls and maintaining exactly-once, in-order delivery of the input to these replicas during failure and recovery. This delicate integration allows the dataflow to handle failures of individual partitions without sacrificing result quality or reducing availability.

In addition to quick fail-over (a direct benefit of replicated computation) Flux also provides automatic on-the-fly recovery that limits disruptions to ongoing dataflow processing. Flux restores accumulated operator state and lost in-flight data for the failed partitions of the dataflow while allowing the processing to continue for unaffected partitions. These features provide the high availability

*This work was supported by NSF grant nos. 0205647, 0208588, a UC MICRO grant, and gifts from Microsft, IBM, and Intel.

SIGMOD 2004 June 13-18, 2004, Paris, France.

necessary for critical applications that not only cannot tolerate inaccurate results and data loss but also have low latency requirements. Moreover, this piecemeal recovery yields a lower mean time to recovery (MTTR) than the straight-forward approach of coordinating replicas of an entire parallel dataflow, thus improving the reliability, or mean time to failure (MTTF), of the system. For restoring lost state, Flux leverages an API, supplied by the operator writer, for extracting and installing the state of operator partitions.

Inspired by Exchange [13], Flux encapsulates the coordination logic for fail-over and recovery into an opaque operator used to compose parallelized dataflows. Flux is a generalization of the Exchange [13], the communication abstraction used to compose partitioned-parallel query plans. This technique of encapsulating the fault-tolerance logic allows Flux to be reused for a wide variety of database operators including stateful ones. This design relieves the burden on the operator writer of repeatedly implementing and verifying critical fault-tolerance logic. By inserting Flux at the communication points within a dataflow, an application developer can make the entire dataflow robust.

In this paper, we describe the Flux design and recovery protocol and illustrate its features. In Section 2, we start by stating our assumptions and outline our basic approach for fault tolerance and high availability. Inspired by process-pairs, in Section 3, we describe how to coordinate replicated single-site dataflows to provide fault tolerance and high availability. Section 4 describes the necessary modifications to Exchange for CQ dataflows and the problems with a naive application of our single-site technique to partitioned parallel dataflows. In Section 5, we build on the techniques in the previous sections to develop Flux's design and its recovery protocol. We also demonstrate the benefits of Flux with experiments using an implementation of Flux in the TelegraphCQ code base. Section 6 summarizes the related work. Section 7 concludes.

2. ENVIRONMENT AND SETUP

In this section, we outline the model for dataflow processing and our basic approach for achieving high availability and fault-tolerance. We also describe the underlying platform, the faults we guard against, and services upon which we rely. Finally, we present a motivating example used throughout the paper.

2.1 Processing Model

A CQ dataflow is a generalization of a pipelined query plan. It is a DAG in which the nodes represent "non-blocking" operators, and the edges represent the direction in which data is processed. A CQ dataflow is usually a portion of a larger end-to-end dataflow. As an example, consider a packet sprayer that feeds data to a CQ dataflow. Imagine the dataflow monitors for intrusions and its output is spooled to an administrator's console. In this example, the larger dataflow includes the applications generating packets and the administrators that receive notifications. In this paper, we only concentrate on the availability of a CQ dataflow and not the end-to-end dataflow nor the entry and exit points. We focus on CQ dataflows that receive input data from a single entry point and return data to clients through a single exit, thereby allowing us to impose a total order on the input and output data. Also, we will only discuss linear dataflows for purposes of exposition, but our techniques extend to arbitrary DAGs. For the cluster-based setting, we assume a partitioned dataflow model where CQ operators are declustered across multiple nodes, and multiple operators are connected in a pipeline [25].

CQ operators process over infinite streams of data that arrive at their inputs. Operators export the Fjord interface [24], `init()` and `processNext()` which are similar to the traditional iterator interfaces [14]. When the operator is invoked via the `processNext()` call, it performs some processing and returns a tuple if available for output. However, unlike `getNext()` of the iterator interface, `processNext()` is non-blocking. The `processNext()` method performs a small amount of work and quickly returns control to the caller, but it need not return data. Operators communicate via queues that buffer intermediate results. The operators can be invoked in a push-based fashion from the source or in a pull-based fashion from the output, or some combination [7, 24].

2.2 Platform Assumptions

In this section, we describe our platform, the types of faults that we handle, and cluster-based services we rely upon external to our mechanism. For this work, we assume a shared-nothing 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. We assume the network layer provides a reliable, in-order point-to-point message delivery protocol, e.g. TCP. Thus, a connection between two operators is modeled as two separate uni-directional FIFO queues, whose contents are lost only when either endpoint fails.

We ignore recurrent deterministic bugs and only consider hardware failures or faults due to "heisenbugs" in the underlying platform. These faults in the underlying runtime system and operating system are caused by unusual timings and data races that arise rarely and are often missed during the quality testing process. When these faults occur, we assume the faulty machine or process is *fail-stop*: the error is immediately detected and the process stops functioning without sending spurious output. Schneider [28] and Gray and Reuter [15] show how to build fail-stop processes. Moreover, since we aim to provide both consistency and availability, we cannot guard against arbitrary network partitions [12].

We rely on a cluster service that allows us to maintain a consistent global view of the dataflow layout and active nodes in the cluster [33]. We call it the *controller*. We rely on it to update group membership to reflect active, dead, and standby nodes, setup and teardown the dataflow, and perform consistent updates to the dataflow structure upon failures. Standard group communication software [6, 33] provides functionality for managing membership and can be used to maintain the dataflow layout. Note, the controller is not a single point of failure, but a uniform view of a highly available service. Such a service usually employs a highly available consensus protocol [22], which ensures controller messages arrive at all cluster nodes in the same order despite failures.

However, we do not rely on the controller to maintain or recover any information about the in-flight data or internal state of operators. The initial state of operators must be made persistent by external means, and we assume it is always available for recovery. Our techniques also do not rely on any stable storage for making the transient dataflow state fault-tolerant. When nodes fail and re-enter the system, they are stateless. Also, link failures are transformed into a failure of one of the endpoints.

2.3 Basic Approach

Our goal is to make the in-flight data and transient operator state fault-tolerant and highly available. In-flight data consists of all tuples in the system from acknowledged input from the source to unacknowledged output to the client. This in-flight data includes intermediate output generated from operators within the dataflow that may be in local buffers or within the network itself.

Inspired by the process-pairs technique [15], our approach provides fault-tolerance and high availability by properly coordinating redundant copies of the dataflow computation. Redundant computation allows quick fail-over and thus gives high availability. We restrict our discussion in this paper to techniques for coordinating two replicas; thus, we can tolerate a single failure between recovery

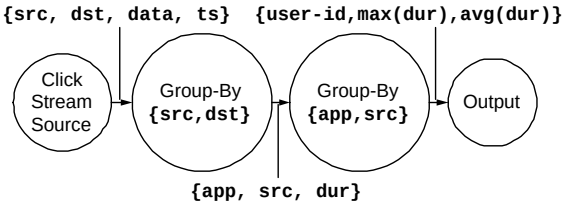


Figure 1: *Example Dataflow*

points. While further degrees of replication are possible with our technique, for most practical purposes, the reliability achieved with pairs is sufficient [15].

We model CQ operators as deterministic state-machines; thus, given the same sequence of input tuples, an operator will produce the same sequence of output tuples. We call this property *sequence-preserving*. Most CQ operators fall within this category, for example, windowed symmetric hash join or windowed group-by aggregates. Section 3.4 describes how to relax this assumption for single-site dataflows.

Our techniques ensure that replicas are kept consistent by properly replicating their input stream during normal processing, upon failure, and after recovery. With the dataflow model, maintaining input ordering is straightforward, given an in-order communication protocol and sequence-preserving operators. However, consistency is difficult to maintain during failure and after recovery because connections can lose in-flight data and operators may not be perfectly synchronized. Thus, our techniques maintain the following two invariants to achieve this goal.

1. Loss-free: no tuples in the input stream sequence are lost.

2. Dup-free: no tuples in the input stream sequence are duplicated.

To maintain these invariants, we introduce intermediate operators that connect existing operators in a replicated dataflow. Between *every* producer-consumer operator pair that communicate via a network connection, we interpose these intermediate operators to coordinate copies of the producer and consumer. Abstractly, the protocol is as follows. To keep track of in-flight tuples, we assign and maintain a sequence number with each tuple. The intermediate operator on the consumer side receives input from its producer, and acknowledges the receipt of input (with the sequence number) to the producer's copy. The intermediate operator at the producer's copy stores in-flight tuples in an internal buffer and ensures our invariants in case the original producer fails. The acknowledgements track the consumer's progress and are used to drain the buffer and filter duplicates. We will show that by composing an existing dataflow using intermediaries that embody this unusual, seemingly asymmetric protocol in various forms, the dataflow can tolerate loss of its pieces and be recovered in piecemeal.

2.4 Motivating Example

As a motivating example, we consider a dataflow that may be used for analyzing network packets flowing through a firewall (or DMZ) for a large organization. Suppose a network administrator wants to track the maximum and average duration of network sessions across this boundary in real time. The sessions may be for various applications, e.g. HTTP, FTP, Gnutella, etc. Further, suppose the administrator wants these statistics on a per source and application basis. Such information may be used, for example, to detect anomalous behavior or identify misbehaving parties. The linear dataflow in Figure 1 performs this computation; we will use this as our example throughout the paper.

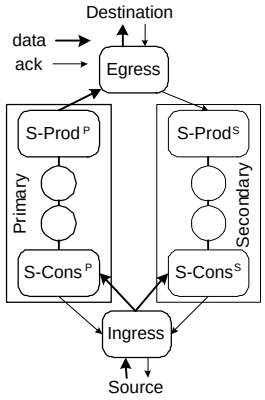


Figure 2: Dataflow Pairs Normal Processing

The data source, the first operator, provides a raw packet stream. To determine session duration, the network session must be reconstructed from a packet stream. Each packet stream tuple has the schema, `(src, dst, data, ts)`, where the source and destination fields determine a unique session. Each tuple also has a timestamp, `ts`, and a `data` field that contains the payload or signals a start or end of session. The second operator is a streaming group-by aggregation that reconstructs each session, determines the application and duration, and outputs a `(app, src, dur)` tuple upon session completion. The third operator is also a streaming group-by aggregation that maintains for each application and source, i.e. `(app, src)`, the maximum and average duration over a user-specified window and emits a new output at a user-specified frequency. The final operator is an output operator.

3. SINGLE-SITE DATAFLOW

Inspired by process-pairs, in this section, we describe how to coordinate replicated, single-site CQ dataflows to provide quick failover and thus high availability. We extend our technique to parallel dataflows in the next section.

For a given CQ dataflow, we introduce additional operators that coordinate replicas of the dataflow during normal processing and automatically perform recovery upon machine failure. We refer to these operators as *boundary* operators because they are interposed at the input and output of the dataflow. These operators encapsulate the coordination and recovery logic so modifications to existing operators are unnecessary. Recovery has two phases: *take-over* and *catch-up*. Immediately after a failure is detected, take-over ensues. During take-over, we adjust the routing of data within the dataflow to allow the remaining replica to continue processing incoming data and delivering results. After take-over, the dataflow is vulnerable: one additional failure would cause the dataflow to stop. Once a standby machine is available, the catch-up phase creates a new replica of the dataflow, making the dataflow once again fault-tolerant.

3.1 Dataflow Pairs

In this section, we describe the normal-case coordination necessary between replicated dataflows to guard against failures.

3.1.1 Components of a Single-Site Dataflow

A single-site CQ dataflow is a DAG of non-blocking operators which are typically in a single thread of control. In our example, the dataflow is a pipeline. We call any such locally connected sequence of operators a *dataflow segment*, as shown in the left portion of Figure 2. Communication to non-local machines occurs only at the top and the bottom of the dataflow segment. When we refer to the

top, we mean the end that delivers output and the bottom is the end that receives input. Likewise, we use "above" to indicate closer to the output, and vice-versa for "below".

In top-down execution, the topmost operator invokes the operators below it recursively through the `processNext()` method, and returns the results through an *egress* operator that forwards results to the destination. At the bottom are operators that receive data from an *ingress* operator. Ingress handles the interface to the network source. The ingress and egress are boundary operators. The circles are operators in the dataflow; in our running example they are the streaming group-by operators.

In this paper, we place the ingress and egress operators on separate machines and assume they are always available. These operators are proxies for the input and output of the dataflow and may have customized interfaces to the external world. Existing techniques, e.g. process pairs, may be used to make these operators highly available, but we do not discuss these further. Our focus is on the availability of the dataflow that processes the incoming data and its interaction with these proxy operators. Thus, in the remaining discussion, we detail ingress and egress alongside the other boundary operators we introduce.

3.1.2 Normal Case Protocol

To make a single-site dataflow fault-tolerant and highly available, we have two copies of the entire dataflow running on separate machines and coordinate the copies through the ingress and egress. Henceforth, we use the superscript P to denote the primary copy of an object, and the superscript S to denote the secondary (see Figure 2). Ingress incorporates the input and forwards it to both dataflows, loosely synchronizing their processing, and egress forwards the output.

The ingress operator incorporates the network input into data structures accessible to the dataflow execution engine, i.e. tuples. Once the ingress operator receives an input and has incorporated it into the dataflow, it sends an acknowledgement (abbreviated as ack) to the source indicating that the input is stable and the source can discard it. For push-based sources that do not process acks, the operator does its best to incorporate the incoming data.

The egress operator does the reverse. It sends the output to the destination and when an ack is received from the destination, the egress operator can discard the output. If the destination does not send acks, result delivery is also just best-effort.

We introduce additional boundary operators called S-Prod and S-Cons at the top (producing) and at the bottom (consuming) end of the dataflow segment respectively. We assume each input tuple is assigned a monotonically increasing input sequence number (ISN) from the source or ingress operator. The ingress operator buffers and forwards each input tuple to both S-Cons^P and S-Cons^S which upon receipt send acks. The S-Cons operators do not forward an incoming tuple to operators above unless an ack for that tuple has been sent to the ingress. Acknowledgments, unless otherwise noted, are just the sequence numbers assigned to tuples. When ingress receives an ack from both replicas for a specific ISN, it drops the corresponding input from its internal buffer. Likewise, both S-Prod operators assign an output sequence number (OSN) to each output and store the output in an internal buffer. Only S-Prod^P forwards the output to the egress operator after which it immediately drops the tuple. Egress sends acks to S-Prod^S for every input received. Once an ack is sent, egress can forward the input to the destination. Once S-Prod^S has received the ack from egress, it discards the output tuple with that OSN from its buffer. Note, acks may arrive before output is produced by the dataflow. S-Prod^S maintains these acks in the buffer until the corresponding output is produced. We will see in Section 5, that this asymmetric, egress protocol is exactly half of the symmetric, Flux protocol.

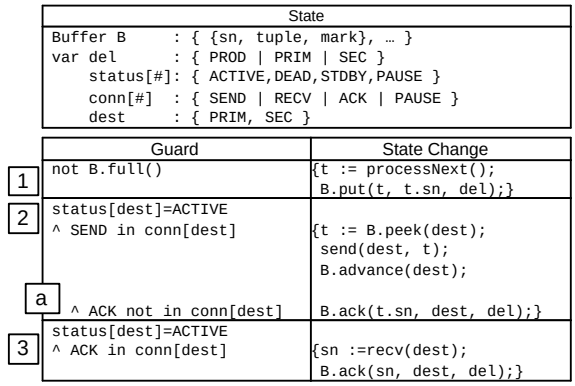


Figure 3: Abstract Producer Specification - Normal Case

The dataflow pairs scheme ensures both dataflows receive the same input sequence and maintains the loss-free and dup-free invariants in the face of failures. The former is true because of our in-order delivery assumption about connections. Next, we show how to maintain the latter by using the internal buffers. The buffer serves as a redundant store for in-flight tuples as well as a duplicate filter during and after recovery. We describe the interface these buffers support and how the boundary operators use them during normal processing and recovery.

3.1.3 Using Buffers

The buffer in the ingress operator stores sequence numbers and associated with those sequence numbers stores tuples and markings. If a sequence number does not have a tuple associated with it, we call it a *dangling* sequence number. The markings indicate the places from which the sequence number was received. A marking can be any combination of `PROD`, `PRIM`, and `SEC`. The `PROD` mark indicates it is from a tuple produced from below. The `PRIM` and `SEC` markings indicate it is from an ack received from the primary and secondary destinations, respectively. The buffer also maintains two cursors: one each for the primary and secondary destinations. The cursors point to the first undelivered tuple to each destination. The buffer in S-Prod is the same, except there is only one destination, so only one cursor exists and only two markings are allowed, `PROD`, `PRIM`. The buffer supports the following methods: `peek(dest)`, `advance(dest)`, `put(tuple,SN,del)`, `ack(SN,dest,del)`, `ack_all(dest,del)`, `reset(dest)`.

The `peek()` method returns the first undelivered tuple for the destination and `advance()` moves the cursor for that destination to the next undelivered tuple. The `put()` method inserts a sequence number, SN, into the buffer if the SN does not exist. Then, it associates a tuple with that sequence number, and marks that tuple as produced. The `ack()` method marks the sequence number as acknowledged by the given destination. The `del` parameter for both `put()` and `ack()` indicates which markings must exist for a sequence number in order to remove it and its associated tuple from the buffer. The `reset()` and `ack_all()` methods are used during take-over and catch-up, so we defer their description to Section 3.2.

Our operators that produce or forward data use this buffer and implement the abstract state-machine specification in Figure 3. The specification contains state variables and actions that can modify the state or produce output. State variables can be scalar or set-valued. The values they can hold are declared within braces, and for set-valued types, these values are separated by bars. Each row in the specification is an action. Each action has a guard, a predicate that must be true, to enable the action. Each enabled action causes

a state change. State change commands surrounded by braces imply the sequence of commands is atomic. That is, either all or no commands are performed, and the action makes no state changes if any command in the sequence fails. If multiple actions are enabled, any one of their state changes may occur. Indented predicates are shorthand for nested conditions; the higher level predicate must also be true. If the nested condition is also true, their corresponding commands are appended to the higher level command sequence. Actions may also have external actions as pre-conditions (see Figure 4). While the abstract producer description only specifies actions related to the output side of the operators, it suffices to illustrate how the buffer is used.

The specification in Figure 3 indicates that any one of three actions may occur in a data forwarding operator like ingress or S-Prod. If there is room in the buffer, the operator gets the next incoming tuple, determines the sequence number, and inserts it using `put()` (action (1)). If a destination is alive and its connection indicates that it should send, then it removes the first undelivered tuple, sends it, and advances the cursor (action (2)). If the connection also is not receiving acks, the tuple's sequence number is immediately acked in the buffer after sending (action (2a)). If the connection indicates that acks should be processed, the next acked sequence number is retrieved and is stored in the buffer (action (3)). Note, if either t or sn is null, the actions make no state changes.

By setting the variables appropriately, we obtain the behavior of our boundary operators that produce data. For the ingress operator, the `dest` variable ranges over both primary and secondary connections. For both connections, the `conn` variable is set to {`SEND,ACK`} to indicate that the connections are used for both sending data and receiving acks. The `del` variable is set to {`PROD, PRIM, SEC`} to indicate that all three markings are necessary for eviction of an entry. For S-Prod,^{P} `conn[PRIM]= {SEND}`; for S-Prod,^{S} `conn[PRIM] = {ACK}`, i.e. data is not forwarded. For the S-Prod operators, there is only one connection to the egress, so `dest` always is set to `PRIM`, and `del` is set to {`PROD, PRIM`}. Thus, for S-Prod,^{P} entries in the buffer are deleted only after having been produced and sent. For S-Prod,^{S} entries are deleted after having been produced and acked. Note, acks from egress may be inserted in the S-Prod,^{S} buffer before their associated tuples are produced, resulting in dangling sequence numbers in the buffer.

The size of the ingress buffer limits the drift of the two replicas; the shorter the buffer the less slack is possible between the two. Since we assume underlying in-order message delivery, the buffers can be implemented as simple queues, and the acks can be sent periodically by S-Cons or egress to indicate the latest tuple received. With this optimization, `ack()` acknowledges every tuple with sequence number $\leq$ SN. This scheme allows us to amortize the overhead of round-trip latencies, but take-over and catch-up protocols must change slightly. We omit these details due to space constraints.

The mean-time-to-failure (MTTF) analysis for the dataflow pair during normal processing is the same as that of process pairs, assuming independent failures. The overall MTTF of the system is $(\text{MTTF}_s)^2/\text{MTTR}_s$ where MTTR_s is the time to recover a single machine, MTTF_s is the MTTF for a single machine, and $\text{MTTR}_s \ll \text{MTTF}_s$ [15]. In our case, MTTR_s consists of both the take-over and catch-up phase. The latter only occurs if a standby is available.

3.2 Take-Over

In this section, we describe the actions involved in the take-over protocol for the boundary operators. We begin by discussing how controller messages are handled and what additional state the boundary operators must maintain. We then describe the protocol and show how it maintains our two invariants after a failure.

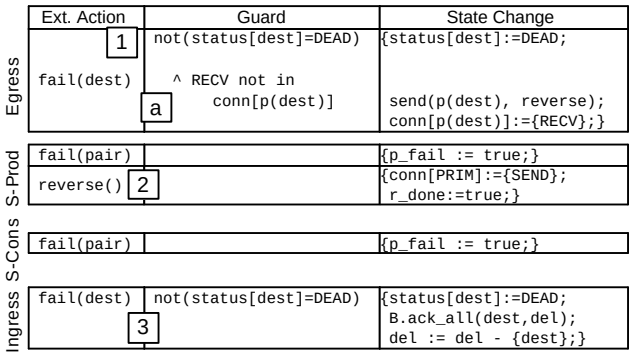


Figure 4: Take-Over Specification

3.2.1 Controller Messages and Operator Modes

As mentioned before, we rely on the controller to the maintain group membership and dataflow layout, and it can be implemented using standard software [6, 33]. Typically, such a service will detect failures via timeout of periodic heartbeats, and inform all active nodes of the failed node through a distributed consensus protocol [22] (i.e. do a view update) . Note, node failures are permanent; a reset node enters the cluster at initial state. When a failure message arrives from the controller (via a view update), we model it as an external action `fail()` invoked on our boundary operators. The controller also sends availability messages (`avail()`) used during catch-up (see Section 3.3). Since the controller can send multiple messages, we enforce that all enabled actions for a controller message complete before actions for the next message can begin.

While controller messages arrive at all machines in same order within all controller commands, there is no ordering guarantee with respect to data routed within a dataflow. For example, a `fail()` message may arrive at the ingress before some tuple t has been forwarded and arrive at a S-Cons much after the tuple t has been received. Thus, our boundary operators must coordinate using messages within the dataflow to perform take-over.

Moreover, our boundary operators must maintain the status of the operator on the other end of all outgoing and incoming connections. Each operator can be in four distinct modes: `ACTIVE`, `DEAD`, `STDBY`, `PAUSE`. In the `ACTIVE` mode, the operator is alive and processing. When it is dead, it is no longer part of the dataflow. We discuss the `STDBY` and `PAUSE` modes in Section 3.3 of catch-up.

3.2.2 Re-routing Upon Failure

Upon failure, we need to make two adjustments to the routing done by our boundary operators to ensure the remaining dataflow segment continues computing and delivering results. First, in the ingress operator, we must adjust the buffer to no longer account for the failed replica. Second, if the primary dataflow segment fails, the secondary S-Prod must forward data to egress. The actions enabled upon a failure and during take-over for our boundary operators are shown in Figure 4. We describe these top-down in our dataflow.

When a failure message arrives at the egress operator, first it simply adjusts the connection status to indicate that the destination is dead (action (1)). If the connection to the remaining replica, indexed by `p(dest)`, is receiving results, nothing else happens. Otherwise, the egress operator marks that connection for receiving tuples. Egress also sends a `reverse` message to the replica (action (1a)). Hence, if the primary dataflow segment fails, egress begins processing results from the secondary. The reverse message is sent along the connection to S-Prod that was used for acks. This message is an indication to S-Prod to begin forwarding tuples instead of processing acks.

When a reverse message arrives at the S-Prod, it simply adjusts the connection state to begin sending to its only connection (action (2)). Using the buffer ensures that no results are lost or duplicated. Once the reverse message arrives, there are no outstanding acks from egress, given our assumption about in-order delivery along connections. At this point, there can be two types of entries in the buffer: unacknowledged tuples, and dangling sequence numbers. The dangling sequence numbers are from received acks for which tuples have yet to be produced. No tuples are lost because all lost in-flight tuples from the primary either remain unacknowledged in the buffer or will eventually be produced, assuming the ingress is fault-tolerant. Moreover, newly produced tuples with already acked sequence numbers will not be resent because the buffer acts as a duplicate filter. Once produced, a tuple is inserted using the `put()` method and is immediately removed if a dangling sequence number for it exists.

When S-Prod starts sending, the buffer ensures the primary's cursor points to the first undelivered tuple in the buffer, i.e. the first unacknowledged tuple. As described in the next section, the S-Prod and S-Cons operators maintain additional state, `p_fail`, used to indicate the completion of the take-over phase.

Upon receiving the failure message, the ingress operator first sets the connection to dead (action (3)). Then, using the `ack_all()` method, it marks all tuples destined for the dead connection as acknowledged. Like `ack()`, `ack_all()` will remove tuples containing all the marks in `del`. Finally, ingress adjusts the `del` variable to only consider the produced marker, `PROD`, and one of `PRIM` or `SEC` for the remaining connection, when evicting tuples. No tuples are lost or duplicated because for the remaining connection, the ingress operator is performing exactly the same actions. Moreover, the buffer does not fill indefinitely after take-over because `del` is modified to ignore the dead connection.

After take-over is complete, the dataflow continues to process and deliver results. But, it is still vulnerable to an additional failure.

3.3 Catch-Up

To make the dataflow fault-tolerant again, we initiate a recovery protocol called catch-up. Figure 5 shows the specification for the catch-up phase. During catch-up, a passive standby dataflow is brought up-to-date with the remaining copy. To do so, we need another non-faulty machine that has the dataflow operators initialized in `STDBY` mode. We assume that the controller arranges such a machine after failure and issues an `avail()` message to indicate its availability. We also assume that the controller initializes the standby machine with operators in their initial state and has the connections properly setup with the boundary operators. Once available, the *state-movement* phase begins in which we transfer state of the active dataflow onto the standby machine. Then in the *fold-in* phase we incorporate the new copy into the overall dataflow. Below, we detail these two phases of catch-up.

3.3.1 State-Movement

We first provide an overview of the state-movement phase. Once a standby dataflow segment is available, state-movement begins. Initially, the S-Cons operator quiesces the dataflow segment and begins state movement. S-Cons leverages a specific API (which we describe shortly) for extracting and installing the state of the dataflow operators. This state is transferred through a StateMover that is local to the machine, but in a separate thread outside the dataflow. The StateMover has a connection to the StateMovers on all machines in the cluster. Once state-movement is complete, we have two consistent copies of the dataflow segment. Next, we detail the specific actions taken in this phase.

The S-Cons operator initiates the state-movement phase when it recognizes that a standby dataflow segment is ready via an `avail()` message from the coordinator. The S-Cons operators on both the active and the standby machines rely on a local, but external StateMover process for transferring state. The S-Cons on the active machine is in `ACTIVE` mode, and the S-Cons on the standby machine is in `STDBY` mode. The active S-Cons operator checks with its S-Prod above to determine if take-over has completed (action (1)). If so, it signals the StateMover process to begin state movement. The StateMovers on the standby and active machines communicate, and, when ready for transferring state, they both signal their respective S-Cons operators with `sm_ready()` (action (2)). Next, the active S-Cons increments its version number and pauses the incoming connection. It also calls `initTrans()` which is invoked on every operator in the dataflow segment to quiesce the operators for movement. Eventually the call reaches the S-Prod above and pauses the operator. Then state transfer begins.

At this point, we note two important items. First, S-Prod pauses

Ext. Action	Guard	State Change
avail(dest)		{status[dest]:=STDBY; conn[dest] := {};}
psync(dest,v) 8 *Egress*	status[dest]=ACTIVE ^ ver[p(dest)] = v	{ver[dest] := v; conn[p(dest)]:={ACK};}
	else	conn[dest]:={PAUSE};}
	status[dest]=STDBY ^ ver[p(dest)] = v	status[dest]:=ACTIVE;} conn[p(dest)]:={RECV}; conn[dest]:={ACK};}
tk_done()	p_fail ^ r_done	{return true;}
sync(ver) 7 *S-Prod*	my_status=STDBY	{p_fail:=r_done:=false; send(psync, ver); my_status:=ACTIVE; conn:={ACK};}
initTrans() 3		{p_status:=my_status; my_status:=PAUSE;}
endTrans()		{my_status:=p_status;}
avail(pair) 1	my_status=ACTIVE ^ p_fail ^ S-Prod.tk_done()	{send(sm,send-req};}
	my_status=STDBY	{send(sm,recv-req);}
sm_ready() 2		{my_ver:=my_ver+1; conn:={PAUSE}; initTrans(); move();}
move() 4 *S-Cons*	my_status=ACTIVE	{st:=getState(); send(sm, st); sm_done();}
	my_status=STDBY	{st:=recv(sm); installState(st); sm_done();}
sm_done() 5		{endTrans(),p_fail:=false; S-Prod.sync(my_ver); conn:={RECV, ACK}; my_status:=ACTIVE; send(csync, my_ver);}
avail(dest)		{status[dest]:=STDBY; conn[dest]:={};}
csync(dest, v) 6 *Ingress*	status[dest]=ACTIVE	{ver[dest]:=v; del:=del+{p(dest)}; B.reset(p(dest));
	^ ver[p(dest)]=v	conn[p(dest)]:={SEND,ACK };}
	status[dest]=STDBY ^ ver[p(dest)]=v	{ver[dest]:=v; status[dest]:=ACTIVE; conn[dest]:={SEND,ACK};}

Figure 5: Catch-Up Specification

to prevent additional state changes from occurring during movement (action (3)). We enforce `not(my_status=PAUSE)` as a guard for every action to stall the dataflow segment completely. (Section 5 shows how to alleviate this stall for parallel dataflows.) Second, we have introduced version numbers for each dataflow segment. The version number tracks which checkpoint of the dataflow state has been transferred. Ingress and egress also maintain the checkpoint versions for the segments at the other end of their connections. During state movement, the version number is copied so the active's and standby's version values match after movement. These checkpoint version numbers are then used to match the synchronization messages sent during the fold-in phase.

In order to transfer the state of dataflow operators, operator developers must implement a specific API. First, we require two methods: `getState()` and `installState()` (action (4)) that extract and marshall, and unmarshall and install the state of the operator, respectively. S-Cons calls these methods on every dataflow operator during state movement. For example, for the group-by from our example, `getState()` extracts the hash table entries that contain intermediate per-group state and marshalls it into machine independent form; `installState()` does the reverse. Even S-Prod implements these methods to transfer its internal buffer. Similarly, we use `initTrans()` to quiesce the operator before transfer and `endTrans()` to restart the operator afterwards.

Once state transfer is finished, the `sm_done()` action is enabled (action (5)). Next, both S-Cons operators restart their ancestors, restart their input connection for receiving and acking data, and enable `sync()` on their downstream S-Prod operator. At this point, the two dataflow copies are consistent with respect to their state, the amount of input processed, and output sent. The fold-in phase restarts the input and output streams of the new replica while maintaining the dup-free and loss-free invariants.

3.3.2 Fold-In

Once state movement is complete both active and standby S-Prod and S-Cons operators begin fold-in by sending synchronization messages to the egress and ingress operators, respectively. The messages mark the point, within the output or input streams of the dataflow segment, at which the two new replicas are consistent. Once the ingress and egress operators receive these messages from the active and standby, the incoming and outgoing connections are restarted and fold-in is complete. We first describe the interaction between the ingress and S-Cons operators, and then describe the interaction between the egress and S-Prod operators.

S-Cons sends a `csync` message with its latest checkpoint version to the ingress operator (action (5)). Note, the `csync` messages are sent along the connection to the ingress for sending acks, flushing all acks to the ingress. Once sent, both S-Cons are active. On the other side, the ingress must properly incorporate both messages and restart the connections to the S-Cons operators.

The ingress operator completes fold-in after both `csync` messages arrive (action (6)). Since the `csync` messages can arrive at the ingress in any order, we must keep track of their arrivals and take actions to avoid skipping or repeating input to the new standby. The `ver` variable maintains the latest checkpoint version for the dataflow at the other end of each connection. Since versions are sent only with csync messages, we use `ver` to indicate whether a `csync` was received from the connection. When `csync` arrives from the active dataflow segment, we are certain that all acks sent prior to state transfer have been received. So, in order to avoid dropping input for the new replica, ingress updates `del` to indicate that acks from both connections are necessary for deletion. Ingress also resets the cursor for the standby through `reset()` which points the cursor to the first of the unacknowledged tuples in the buffer. Reset also ensures the markings for both connections match on each entry and removes entries with all three or no markings. Note, we are careful not to restart the connection to the standby segment unless the `csync` from the active dataflow segment has arrived. Otherwise, ingress might duplicate previously consumed tuples to the new replica. Fold-in is complete when it activates the connection to the new replica.

Returning to the top of the dataflow segment, once S-Prod operators receive the `sync()` from the S-Cons below, they emit a `psync` with the latest checkpoint version along the forward connection to egress (action (7)). After this message is sent, both S-Prod are active, and only fold-in for the egress remains to be discussed.

At the egress operator, we must ensure that it does not forward acks for any in-flight data sent before movement began and that it does not miss sending acks for any tuples sent after movement completed (action (8)). If the `psync` arrives from the active first, we are careful to pause the connection until the second `psync` arrives. Otherwise, egress would miss sending acks for tuples received after state-movement but before the standby `psync` arrives. If `psync` arrives from the standby first, we are careful not to activate the connection for acks; otherwise, egress would forward acks for tuples sent before state-movement. The second `psync` causes both connections to be activated. Once egress starts sending acks to the new replica, fold-in is complete. Once fold-in is completed at both ends, catch-up is finished and the dataflow is fault-tolerant.

3.4 Conclusion - Dataflow Pairs

There are two properties of this protocol that we want to highlight. First, with some more modifications, we can make the catch-up protocol idempotent. That is, even if the standby fails during movement, the dataflow will continue to process correctly and attempt catch-up again. Idempotency can be extremely useful. For example, imagine an administrator wants to migrate the dataflow to a machine with an untested upgraded OS. With such a property, she can simply terminate a replica, bring up a standby with the new OS, and if the standby fails, revert back to the old OS. We sketch the needed changes for achieving idempotent catch-up. On a failure during movement, the S-Cons operator completes the protocol as if movement did finish. But, multiple failures can cause the checkpoint version numbers on the outgoing (incoming) connections of ingress (egress) to drift apart. Thus, we must modify the ingress and egress `psync()`, `csync()`, and `fail()` actions to track this gap and properly restart connections when the versions match.

Second, note the entire protocol only makes use of sequence numbers as unique identifiers; we never take advantage of their order, except in the optimizations for amortizing the cost of round-trip latencies. Some operators exhibit external behavior that depends on inputs outside of the scope of the dataflow. For example, the output order of XJoin [31] depends upon the prevailing memory pressures. If we ignore these optimizations, we can accommodate dataflow segments that include non-deterministic but *set-preserving* operators such as XJoin. That is, given the same *set* of input tuples, the operator will produce the same *set* of output tuples. In this case, instead of generating new OSNs, we require that outputs have a unique key which are used in place of sequence numbers. We outline methods for maintaining such keys in Section 4.1.

4. PARALLEL DATAFLOW

Parallelizing a CQ dataflow across a cluster of workstations is a cost-effective method for scaling high-throughput applications implemented with CQ dataflows. For example, in our monitoring scenario, one can imagine having thousands of simultaneous sessions and thousands of sources. Moreover, the statistics (e.g. max) collected may range over some sizeable window of history. To keep

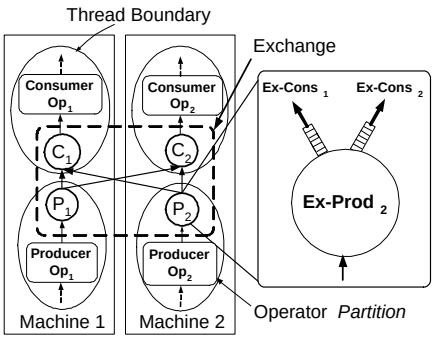


Figure 6: ***Exchange Architecture***

up with high-throughput input rates and maintain low-latencies, the dataflow can be scaled by partitioning it across a cluster.

On a cluster, a CQ dataflow is a collection of dataflow segments (one or more per machine). Individual operators are parallelized by partitioning their input and processing across the cluster, a technique called partitioned parallelism. When the partitions of an operator need to communicate to non-local partitions of the next operator in the chain, the communication occurs via the Exchange[13]. In Section 4.1, we describe Exchange and the necessary extensions for use in a partitioned parallel CQ dataflow consisting of sequence-preserving operators that *require* their input to be in arrival order.

In this configuration, a scheme that naively applies the dataflow pairs technique without accounting for the cross-machine communication within a parallel dataflow quickly becomes unreliable. In Section 4.2, we show that this naive approach, called *cluster pairs*, leads to a mean-time-to-failure (MTTF) that falls off quadratically with the number of machines. Moreover, the parallel dataflow must stall during recovery, thereby reducing the availability of the system. Instead, embedding coordination and recovery logic within the Exchange speeds up the MTTR thereby improving both availability and MTTF. The improved MTTF falls off linearly with the number of machines. In Section 5, we describe the Flux design which achieves this MTTF.

4.1 Exchange

In a parallel database, an Exchange [13] is used to connect a producer-consumer operator pair in which the producer's output must be repartitioned for the consumer. In our running example, a viable way to parallelize the dataflow is to partition the group-bys' state and input based on `(src, dst)` at the first level, and `(app, src)` at the second. The Exchange ensures proper routing of data between the partitioned instances of these operators. Exchange is composed of two intermediate operators, Ex-Cons and Ex-Prod (see Figure 6). Ex-Prod encapsulates the routing logic; it forwards output from a producer partition to the appropriate consumer partition based on the output's content. In our example, if the partitioning was hash-based, Ex-Prod would compute a hash on `(app, src)` to determine the destination. Since, any producer partition can generate output destined for any consumer partition, each Ex-Prod instance is connected to each Ex-Cons instance. Typically, Ex-Cons and Ex-Prod are scheduled independently in separate threads and support the iterator interface. Ex-Cons merges the streams from the incoming connections to a consumer instance. Since Exchange encapsulates the logic needed for parallelism, the operator writer can write relational operators while being agnostic to their use in a parallel or single-site setting.

We make two modifications to Exchange for use in CQ dataflows. First, to allow a combination of push and pull processing, Ex-Prod and Ex-Cons must support the non-blocking Fjord interfaces [24]. Second, Ex-Cons must be order-preserving. Some CQ operators, like a sliding window group-by whose window slides with every new input, require that its input data arrive in sequential order. In the parallel setting, since the input stream is partitioned, the input will arrive in some interleaved order at an Ex-Cons. The streams output by the individual Ex-Prod instances are, however, in sequential order. An Ex-Cons instance can recover this order by merging its input data using their sequence numbers. We can also modify Ex-Cons to support operators that relax this ordering constraint.

An important related issue is how to maintain sequence numbers as tuples are processed through the dataflow. We cannot simply generate new output sequence numbers (OSNs) at Ex-Prod otherwise the ordering across Ex-Prod instances would be lost. Instead, we must keep the original ISN intact to reconstruct the order at the consumer side. For operators that perform one-to-one transformations, the operators just need to keep the input tuple's sequence number (SN) intact. For one-to-many transformations, the output SN can be a compound key consisting of the input SN, and another value that uniquely orders all tuples generated from that input. For example, for a symmetric join, a concatenation of the SNs from its two input streams will suffice. For many-to-one transformations like windowed aggregates, the largest SN of the input that produced the output will suffice. It is the task of operator developers to generate correct SNs.

4.2 Naive Solution: Cluster Pairs

In this section we describe how to make a parallel dataflow highly available and fault-tolerant in a straightforward manner using the technique in Section 3. Assume we still have a single ingress and egress operator. Also assume we use partitioned parallelism for the entire dataflow. Each machine in the cluster is executing a single-site dataflow except the operators only process a partition of the input and repartition midway if needed.

A naive scheme for parallel fault-tolerance would be to apply the ideas of the previous section only to the operator partitions on each machine that communicate with ingress and egress. Using this technique, we can devote half of the machines in a cluster for the primary partitions of the dataflow and the other half for the secondary, with each machine having an associated pair (copy). We call this scheme *cluster pairs*. We refer to the set of machines running either the primary partitions or the secondary partitions as a replica set. With cluster pairs, an operator partition and its copy are not processing input in lock-step; thus, when a machine fails, the copies may be inconsistent. Since partitioned operators communicate via Exchange, we may lose in-flight data on failure making it impossible to reconcile this inconsistency. Thus, a single machine failure in either the primary or secondary replica set renders the entire dataflow replica set useless.

To see why, from our example, imagine a machine in the primary replica set failed with a partition of the two group-by operators on it. Lets call the lower streaming group-by operator G_1 and the upper G_2. At the time of failure, a secondary partition of G_1 may have already produced, say a hundred tuples, that its primary copy was just about to send. Now, if we recovered the state from the remaining secondary partition, all the primary operator partitions of G_2 that relied on the failed primary partition will never receive those hundred tuples. From that point forward, the primary partitions will be inconsistent with their replicas. Without accounting for the communication at the Exchange points, we must recover the state of the entire parallel dataflow across all the machines in the primary replica set to maintain consistency and correctness.

The MTTF for this naive technique is the same as the process-pair approach $(\text{MTTF}_c)^2/\text{MTTR}_c$ where MTTF_c and MTTR_c are for a replica set. Since the replica set is partitioned over $N/2$

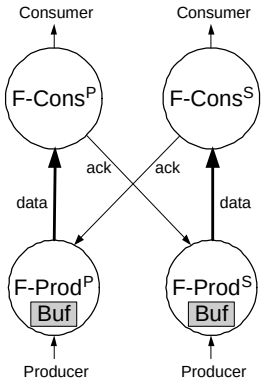


Figure 7: *Flux design and normal case protocol.*

machines, $\text{MTTF}_c = 2\text{MTTF}_s/N$. Thus, the overall MTTF is $4(\text{MTTF}_s)^2/(N^2\text{MTTR}_c)$, a quadratic drop off with N.

The catch-up phase is the bulk of our recovery time, it is the determining factor in this equation. Depending on the available bandwidth during catch-up, $\text{MTTR}_s \leq \text{MTTR}_c \leq N \times \text{MTTR}_s$. More importantly, we need all $N/2$ machines available before catch-up completes and, during catch-up, the entire parallel dataflow is stalled. This stall may lead to unacceptably long response times and perhaps even dropped input.

Clearly, the reliability of the cluster pairs technique does not scale well with the number of machines, nor does it provide the high-availability we want during recovery for our critical, high-throughput dataflow applications. If we had a technique that properly coordinated input to operator partitions at the Exchange points, then we could build parallel dataflows that tolerate the loss of individual partitions. Also, we would only need to recover the state of the failed partition, resulting in improved reliability and availability. In our configuration above, the MTTF for such a scheme would be the time for any paired nodes to fail. Since there are $N/2$ paired nodes, and the MTTF for any pair is $(\text{MTTF}_s)^2/\text{MTTR}_s$, overall the MTTF would be $2(\text{MTTF}_s)^2/(N\text{MTTR}_s)$.

To achieve this improved MTTF, we must modify the Exchange to coordinate operator partition replicas and properly adjust the routing after failures and recovery.

5. PARTITION PAIRS

Our analysis in the previous section shows that without proper coordination of operator partitions at their communication points, recovery is inefficient, leading to reduced reliability and availability. In this section, we build on the protocols for the single-site case and show how to coordinate operator partitions by modifying the Exchange. We show how to maintain the loss-free and dup-free invariants for operator partitions rather than entire dataflows. This design will permit us to recover the dataflow piecemeal and allow the processing for the unperturbed parts of the dataflow to continue, thereby improving both reliability and availability.

Our new operator, Flux, has the same architecture as Exchange; its constituent operators are called F-Cons and F-Prod. For each F-Cons instance in the primary dataflow, there is a corresponding F-Cons instance in the secondary dataflow, and likewise for the F-Prod instances. We call any such pair of instances *partition pairs*. The F-Cons protocol is similar to egress' during normal processing and take-over, and similar to S-Cons during catch-up. F-Prod 's protocol is similar to S-Prod during normal case and recovery. During normal processing, each F-Cons instance acks received input from one of its F-Prod instances to its *dual* F-Prod in the replicated dataflow (see Figure 7). We assume Flux uses the order-preserving variant of Exchange to ensure that the input is consumed in the same order for both partition replicas. F-Prod is responsible for routing output to its primary consumers and incorporating acks from the secondary consumers into its buffer.

There are few salient differences between the Flux protocol and the cluster pairs and dataflow pairs schemes. First, the Flux normal case protocol is symmetric between the two dataflows. This property makes Flux easier to implement and test because it reduces the state space of possible failure modes and therefore the number of cases to verify. In the rest of this paper, we artificially distinguish between the primary and secondary versions of the F-Cons and F-Prod. From the point of view of an operator, we use the adjective primary to mean within the same dataflow and secondary to mean within the dual dataflow. Second, Flux handles both multiple producers and multiple consumers. In a partitioned parallel dataflow these producers and consumers are partitions of dataflow operators. Finally, the instances of F-Prod or F-Cons (and operators in their corresponding dataflow segments) are free to be placed on any machine in a cluster as long as they fail independently. This flexibility is useful for administrative and load-balancing purposes. The independence requirement leads to at least one constraint: no two replicas of a partition are on the same machine.

In this section, we describe the modifications necessary to the previous normal-case and recovery protocols to accommodate all-to-all communication between partitioned operators. Since there may be many such communication points in a dataflow, recovery proceeds bottom-up, recovering one level at a time. We have already shown the base cases for the entry and exit points, and now we will show the inductive step at the Exchange points.

5.1 Flux Normal Case

We specify the normal case forwarding, buffering, and acking protocol Flux uses to guard against unexpected failures. F-Cons behaves the same as egress in Figure 2 except that it manages multiple connections for multiple producers. For each tuple received from a connection to a primary F-Prod, it sends an ack of the tuple's sequence number to the corresponding secondary F-Cons, before considering the tuple for any further processing. Meanwhile, for each destination, an F-Prod instance obeys the same abstract, normal-case specification for producers shown in Figure 3. The actions remain the same, but the state size increases. It maintains one set of state variables for each consumer partition pair. We use a subscript, i, to denote the variable associated with partition i.

Unlike the ingress operator of Figure 2, however, F-Prod only forwards data to the primary partition, $\texttt{conn}_i\texttt{[PRIM]} = \{\texttt{SEND}\}$, and only processes acks from the secondary, $\texttt{conn}_i\texttt{[SEC]} = \{\texttt{ACK}\}$. Finally, once a tuple has been produced, sent to the primary, and acked by the secondary, it is evicted from the buffer, i.e. $\texttt{del}_i = \{\texttt{PROD, PRIM, SEC}\}$.

Since F-Prod does not remove any tuples until an ack has been received, all in-flight and undelivered tuples from F-Prod 's replica are in its buffers or will eventually be produced. Thus, this scheme ensures the loss-free invariant with up to a single failure per partition pair. In the next section, we describe the take-over protocol which allows the dataflow to continue processing after failures and ensures that no tuples will be duplicated to consumer instances.

5.2 Flux Take-Over

Take-over ensures that regardless of the number of machine failures, as long as only one replica of each partition pair fails, the dataflow will continue to process incoming data and deliver results. Since the the normal case protocol is symmetric, there are only four distinct configurations in which a particular F-Prod,^{P} F-Prod,^{S} F-Cons,^{P} F-Cons^S quartet can survive after failures. Without loss of generality, these cases are: F-Cons^S fails, F-Prod^S fails,

F-Cons^S and F-Prod^S fail, or F-Cons^P and F-Prod^S and fail. We describe the take-over actions to handle these cases.

For F-Cons, the take-over specification is exactly the same as the one for egress, except the `fail` message now specifies exactly which partition, i, and copy of F-Prod failed. If the primary fails, F-Cons sends a `reverse` message to the secondary and begins receiving data from the secondary. Like S-Cons, it also notes using `p_fail` if its replica failed, because it is responsible for catch-up, as described in the next section.

For F-Prod, take-over is a combination of the ingress actions and S-Prod actions. Like ingress, when it detects a failure for a consumer instance, it marks all unacked sequence numbers in the corresponding buffer for the failed copy using `ack_all(dest, del)`. In this case, in addition to removing entries with all three marks, this method also removes entries with only the `dest` mark, leaving only entries relevant to the remaining consumer partition. So, if the secondary consumer fails, the method will remove all dangling sequence numbers from the secondary, and the buffer only will contain undelivered tuples for the primary. Or, if the primary fails, the buffer may contain tuples not yet acked by the secondary or dangling sequence numbers from the secondary. Moreover, F-Prod also adjusts `del` to ignore the failed partition during the processing between take-over and catch-up.

Like S-Prod, F-Prod notes if its own replica failed and handles `reverse` messages from secondary consumer partitions. The action enabled in this case is different (replaces Figure 4-(2)):

`reverse(i,SEC)`		`{conn`$_i$`[SEC]:={SEND}; r_done`$_i$`:=true;` `del`$_i$`:=del`$_i$`+{SEC};}`

This state change allows F-Prod to properly forward to both consumers partitions if its replica fails and to just the secondary consumer if the primary consumer partition i also fails.

After takeover, we observe that the Flux protocol maintains the loss-free and dup-free invariants by noting its similarity to the protocol on the egress side. In the egress case, we assumed the egress was fault-tolerant, and in this case, F-Cons is fault-tolerant because it is replicated. If F-Cons^S fails or F-Cons^S and F-Prod^S fail, the failed partitions are ignored and buffer entries modified accordingly. If F-Prod^S fails or F-Prod^S and F-Cons^P fail, a `reverse` message is sent and once received F-Prod^P feeds the remaining consumer(s). In the latter cases, the buffer filters duplicates. In all cases, the remaining partitions in both dataflows continue processing. For the interested reader, a straight-forward case by case anaylsis similar to Section 3.2 shows our invariants hold.

5.3 Flux Catch-Up

The catch-up phase for a partition pair is similar to one described for a single-site dataflow. In this section, we detail the differences. For the purposes of exposition, we assume no failures occur during catch-up of a single partition. Achieving the idempotency property in this case is akin to that in Section 3.4.

Once the a newly reset node is available or a standby machine is available, the catch-up phase ensues. Each failed dataflow segment (a partition with its Flux operators) is recovered individually, bottom-up. F-Cons initiates catch-up when it recognizes that catch-up for the previous level is complete and that take-over is complete for its downstream F-Prod (or S-Prod). Like S-Cons, it stalls the operators within its dataflow segment, and transfers state through the StateMover. Once finished, synchronization messages are broadcast to all primary and secondary consumer partitions at the top of the segment and primary and secondary producer partitions at the bottom to fold in the new partition replica.

There are a few differences between catch-up in this case and the single-site case which we outline first. First, before state-movement begins, F-Cons cannot just stall the incoming connections by setting them to `PAUSE`. Thus, F-Cons pauses the primary incoming connections through a distributed protocol that stalls the outgoing connections from all its primary F-Prods. Second, when the contents of the buffer in F-Prod are installed at the standby, all markings in entries must be reversed and cursor positions swapped. We need to do this reversal because the primary and secondary destinations are swapped for the new replica. Finally, the fold-in synchronization messages received at F-Prod and F-Cons are handled slightly differently from the ingress and egress operators. Below, we detail the distributed pausing protocol and fold-in.

5.3.1 State-Movement

When F-Cons begins state-movement, all of its primary and secondary producer instances are finished with catch-up. Thus, it is in the first survival scenario; F-Cons^S has failed. If the remaining F-Cons^P just pauses the connection locally and copies over its state (see Figure 5-(2)), F-Prod^P will not properly account for the data in-flight to F-Cons^P immediately after the pause. Those tuples no longer exist in F-Prod^P's buffer because it is not receiving or accounting for acks. However, acks for those in-flight tuples will arrive after catch-up at F-Prod^P via the new F-Cons^S from F-Prod^S. And these acks would remain in the buffer indefinitely.

Hence, we must ensure there are no in-flight data before state-movement begins. To do so, F-Cons^P sends a `pause` message to all of its producers, which upon receipt pause the outgoing connection and enqueue an ack for the pause on the outgoing connection. Once F-Cons^P receives all the incoming pause-acks, it can begin state transfer. A slight complication arises if F-Cons is order preserving. Since a pause-ack can arrive at anytime, it may prevent F-Cons from merging and consuming incoming tuples from other un-stalled incoming connections. This deadlock occurs because when merging in order, inputs from all partitions are necessary to select in the next tuple in line. Thus, F-Cons must buffer the in-flight tuples internally in order to drain the network and receive all pause acks. Of course, all buffered tuples still need to be acked to F-Prod^S. State transfer is accomplished through the StateMover and is the same as the single-site case.

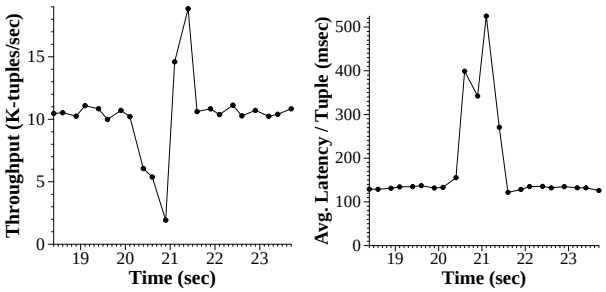
5.3.2 Fold-in

After state-movement, the two partition copies are consistent, and we must restart all connections to the new replica without losing or duplicating input. We first discuss how F-Prod folds in a new F-Cons replica, then describe how F-Cons folds in a new F-Prod.

Both F-Cons^P and the new F-Cons^S broadcast to all producer partitions a `csync` message with the corresponding checkpoint version number. The actions taken by F-Prod are now slightly different than ingress (replaces Figure 5-(6)):

`csync` `(dest,v)`		`{ver[dest]:=v;`
	`status[dest]=ACTIVE`	`del += p(dest);` `B.reset(p(dest));`
	`∧ dest=PRIM`	`conn[PRIM]:={SEND};`
	`∧ ver[SEC]=v`	`conn[SEC]:={ACK};}`
	`∧ dest=SEC` `∧ ver[PRIM]=v`	`conn[PRIM]:={SEND};}`
	`status[dest]=STDBY`	`status[dest]:=ACTIVE;`
	`∧ dest=PRIM` `∧ ver[SEC]=v`	`conn[PRIM]:={SEND};}`
	`∧ dest=SEC` `∧ ver[PRIM]=v`	`conn[SEC]:={ACK};}`

Like the ingress, if a `csync` arrives from the active connection, F-Prod resets the buffer and modifies `del` to account for the new replica. Otherwise, further processing of data (or acks) to (or from) the active might evict tuples in the buffer intended for the



(a) Throughput (b) Avg. Latency

Figure 8: Performance During Recovery

new replica. Note, during state movement, the primary forwarding connection is paused for both F-Prod replicas. If the primary connection is connected to the active replica and a `csync` arrives, the connection immediately is unpaused. If the primary connection goes to a standby replica, the connection is restarted only after the `csync` from the secondary has arrived. This allows F-Prod^S to consume all in-flight acks sent before the movement. Otherwise, F-Prod^S might duplicate tuples consumed by F-Cons^P to the new F-Cons^S. In any case, we start the standby connection only after both syncs arrive.

At the top of the dataflow segment, like S-Prod, both F-Prod operators broadcast a `psync` to all remaining F-Cons. Unlike S-Prod, however, F-Prod can have either one or both F-Cons remaining. F-Prod still broadcasts the `psync` but if it is forwarding data to its secondary, it stops immediately and begins processing acks instead, i.e. `conn[SEC]:={ACK}`. The F-Cons in the next level also must handle the `psync` messages differently according the following action (replacing Figure 5-(8)):

`psync (dest,v)`		`{ver[dest]:=v; status[dest]:=ACTIVE;`
	`dest=SEC ∧ ver[PRIM]=v`	`conn[SEC]:={ACK}; conn[PRIM]:={RECV};}`
	`dest=PRIM`	`conn[PRIM]:={PAUSE};`
	`∧ ver[SEC]=v`	`conn[SEC]:={ACK}; conn[PRIM]:={RECV};}`

To avoid missing acks or sending redundant acks, F-Cons cannot activate a connection upon receiving a `psync` unless the corresponding `psync` from the replica has arrived. Thus, if a `psync` arrives from the primary connection first, then we must pause that connection until the second `psync` has arrived. Otherwise, F-Cons might miss acking tuples received after state-movement. If a `psync` arrives from the secondary first, F-Cons cannot start sending acks unless the `psync` arrives from the primary. Otherwise, F-Cons might resend acks for tuples sent before state-movement. Unlike egress, after both `psyncs` arrive, F-Cons^P acks F-Prod^S and receives data from F-Prod^P regardless of which was active before catch-up.

5.4 Experiment

In this section, we illustrate the benefits of our design by examining the performance of a parallel implementation of our example dataflow during failure and recovery. We implemented a streaming hash-based group-by aggregation operator (2K lines of C code), our boundry operators, and Flux (11K lines of C code) within the TelegraphCQ open-source code base.

In this experiment, we partition this dataflow across four machines in a cluster and place the ingress and egress operators on a separate fifth machine. The ingress has a 400K duplicate elimination buffer. We insert a Flux after the first group-by operator to repartition its output on `(app, src)`. At startup, we place a partition of each operator on each of the four machines, numbered 0 to 3, and replicate them using a chained declustering strategy [16]. That is, each primary partition has its replica on the next machine, and the last partition has its replica on the first. For example, the primary copy of partition 3 of the first group-by is on machine 3 and its replica is on machine 0. In this configuration, when a single machine fails all four survival scenarios occur in different partitions. At startup time, we introduce a standby machine with operators in their initial state. We have not implemented the controller, because it involves well known techniques implemented by standard cluster management software [33]. We simulate failure by killing the TelegraphCQ process on one of those machines, which causes connections to that machine to close and raise an error. Each machine has a Pentium III 1.4 GHz CPU, 512MB of RAM and is connected to a 100mpbs switch.

For the purposes of the experiment, to approximate the workload of a high-throughput network monitoring workload, our ingress operator generates sequentially numbered session start and end events as fast as possible. There are 10K unique `(app,src)` values (uniformly distributed), 100K unique `(src,dst)` pairs. The second group-by outputs running statistics every other update.

With this setup, Figure 8 shows the total output rate and average latency per tuple at the egress. In this experiment, the network is the bottleneck. At $t = 20$ sec, when the experiment reaches steady state, we kill one of the four machines. The throughput remains steady for about quarter of a second, and then suddenly drops. The drop occurs because during state movement, the partition being recovered is stalled and eventually causes all downstream partitions to also stall. In this experiment, about 8.5MB of state was transferred in 941 msec. Once catch-up is finished, at about $t = 21$ seconds, we observe a sudden spike in througput. This spike occurs because during movement, all the queues to the unaffected partitions are filled, and ready to be processed once catch-up completes. Around the same time, figure 8(b) shows an increase in latency because the input and in-flight data are buffered during movement. Then the output rate and average latency settle down to normal. During this entire experiment, the input rate at the ingress stayed at a constant 42K tuples/sec with no data dropped. This experiment illustrates that with piecemeal recovery and sufficient buffering (400K), we can effectively mask the effects of machine failures.

To understand the overheads of Flux, we added just enough CPU processing to the lower-level group-by to make it the bottleneck. In this configuration, for a single parallel dataflow, the input rate was 82K tuples/sec, for cluster pairs, 40K tuples/sec, and for Flux, 36K tuples/sec (10% slower than cluster pairs). Additional processing would only reduce the Flux overhead relative to the others.

6. RELATED WORK

There is a plethora of work on fault-tolerance, availability, and recovery, but the most closely related work encompasses mechanisms for making generalized computations fault-tolerant. In contrast, our work focuses on a narrower but still generally useful style of computation: CQ dataflows.

The replicated state-machine approach [21] coordinates redundant computation to provide protection against faults in a distributed environment. Schneider [29] provides a survey of state-machine based approaches. The critical step in this approach is to reach consensus among the replicas for a consistent view of the input sequence. The Paxos [22] algorithm is the most fault-tolerant method for reaching distributed consensus. Process-pairs [15] is similar because it coordinates two processes, but it differs because a single leader determines input order. Persistent queues [4] are an abstraction that make messages persistent across failures. They are too

heavyweight for our scenario because they provide transactional semantics and depend on stable storage. These schemes are black-box techniques that address a client-sever model rather than a chain of computations, one feeding the next. They do not exploit the structure of parallel CQ dataflows to provide improved reliability and availability.

There are a number of disk-based, checkpoint and replay schemes for message passing systems. The authors in [11] survey these methods. The Phoenix project [23] leverages these techniques to provide persistent COM components. These roll-back recovery techniques provide reliability, but not the high availability needed for critical data-streaming applications.

The Isis [5], Horus, Ensemble [6], and Spread [1] projects are generic group membership and communication toolkits for building highly available applications. Isis maintains group membership and provides the application programmer reliable communication like group broadcast or atomic broadcast. Horus and Ensemble are extensible systems in which programmer can layer these reliable communication primitives as necessary for his or her application. Spread provides similar abstractions for wide-area networks. Such toolkits can be used to implement the controller in our system. They have also been used to perform efficient, eager database replication [19]. But, their primitives offer semantics different than that necessary for partitioned parallel dataflow computation.

Finally, we list recent work on availability for CQ dataflows. Our work in [30] describes mechanisms for extending Exchange to provide load-balancing dataflows. Work in the Aurora system [18] describes techniques for building highly available, wide-area CQ dataflows with stateless operators. In contrast, our techniques handle more general, stateful operators and address parallel dataflows.

7. CONCLUSION

In this paper, we show how to achieve high availability and fault-tolerance for critical, long-running, parallel dataflows. Our main contribution is a technique for coordinating replicas of operator partitions within a larger parallel dataflow. It is a delicate combination of partitioned parallelism and process pairs. Our technique is more reliable and more available than the straightforward cluster pairs approach. Our scheme provides online recovery without stalling the ongoing dataflow computation because it allows for recovering the dataflow in piecemeal. The protocols we describe are in encapsulated in an opaque dataflow operator called Flux. Thus, an application developer can reuse Flux with a variety of operators to make existing, brittle dataflows more robust. We believe integrating load-balancing mechanisms into Flux is a necessary next step for providing performance availability [30].

8. REFERENCES

[1] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. Technical Report CNDS-98-4, Johns Hopkins, 1998.

[2] T. Anderson, D. Culler, and D. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, Feb. 1995.

[3] J. Baulier, S. Blott, H. Korth, and A. Silberschatz. A Database System for Real-Time Event Aggregation in Telecommunication. 1998.

[4] P. A. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *SIGMOD*, 1990.

[5] K. Birman et al. ISIS: A System For Fault-Tolerant Distributed Computing. Technical Report TR86-744, Cornell, 1986.

[6] K. Birman et al. The Horus and Ensemble Projects: Accomplishments and Limitations. Technical Report TR99-1774, Cornell, 1999.

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

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

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

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

[11] E. Elnozahy, D. Johnson, and Y. Wang. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. Technical Report CMU-CS-96-181, CMU, 1996.

[12] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 2002.

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

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

[15] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques*. Kaufmann, 1993.

[16] H. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *ICDE*, 1990.

[17] S. Hvasshovd et al. The ClustRa Telecom Database. In *VLDB*, 1995.

[18] J. Hwang et al. A Comparison of Stream-Oriented High-Availability Algorithms. Technical Report CS-03-17, Brown, 2003.

[19] B. Kemme and G. Alonso. Don't Be Lazy, Be Consistent. In *VLDB*, 2000.

[20] C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. *IEEE Symposium on Security and Privacy*, May 2002.

[21] L. Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 1978.

[22] B. Lampson. The ABCD's of Paxos. In *PODC*, Aug. 2001.

[23] D. Lomet and R. Barga. Phoenix Project: Fault Tolerant Applications. *SIGMOD Record*, June 2002.

[24] S. Madden and M. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *ICDE*, 2002.

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

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

[27] V. Paxon. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 1999.

[28] F. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. *Transactions on Computer Systems*, May 1984.

[29] F. Schneider. Implementing Fault-Tolerant Services Using the State-Machine Approach: A Tutorial. *Computing Surveys*, Dec. 1990.

[30] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2003.

[31] T. Urhan and M. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, June 2000.

[32] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *ACSAC*, 2003.

[33] W. Vogels et al. The Design and Architecture of the Microsoft Cluster Service. In *FTCS*, 1998.