Megaphone: Latency-conscious state migration for distributed streaming dataflows

Moritz Hoffmann
Andrea Lattuada
Frank McSherry
Vasiliki Kalavri
John Liagouris
Timothy Roscoe
Systems Group, ETH Zurich
first.last@inf.ethz.ch

ABSTRACT

We design and implement Megaphone, a data migration mechanism for stateful distributed dataflow engines with latency objectives. When compared to existing migration mechanisms, Megaphone has the following differentiating characteristics: (i) migrations can be subdivided to a configurable granularity to avoid latency spikes, and (ii) migrations can be prepared ahead of time to avoid runtime coordination. Megaphone is implemented as a library on an unmodified timely dataflow implementation, and provides an operator interface compatible with its existing APIs. We evaluate Megaphone on established benchmarks with varying amounts of state and observe that compared to naïve approaches Megaphone reduces service latencies during reconfiguration by orders of magnitude without significantly increasing steady-state overhead.

PVLDB Reference Format:
Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *PVLDB*, 12(9): 1002–1015, 2019.
DOI: https://doi.org/10.14778/3329772.3329777

1 Introduction

Distributed stream processing jobs are long-running dataflows that continually ingest data from sources with dynamic rates and must produce timely results under variable workload conditions [1, 3].

To satisfy latency and availability requirements, modern stream processors support *consistent online reconfiguration*, in which they update parts of a dataflow computation without disrupting its execution or affecting its correctness. Such reconfiguration is required during **rescaling** to handle increased input rates or reduce operational costs [15, 16], to provide **performance isolation** across different dataflows by dynamically scheduling queries to available workers, to allow **code updates** to fix bugs or improve business logic [7, 9], and to enable **runtime optimizations** like execution plan switching [30] and straggler and skew mitigation [14].

Streaming dataflow operators are often stateful, partitioned across workers by key, and their reconfiguration requires *state migration*: intermediate results and data structures must be moved from one set of workers to another, often across a network. Existing state migration mechanisms for stream processors either pause and resume parts of the dataflow (as in Flink [10], Dhalion [16], and SEEP [15]) or launch new dataflows alongside the old configuration (as for example in ChronoStream [28] and Gloss [24]). In both cases state moves "all-at-once," with either high latency or resource usage during the migration.

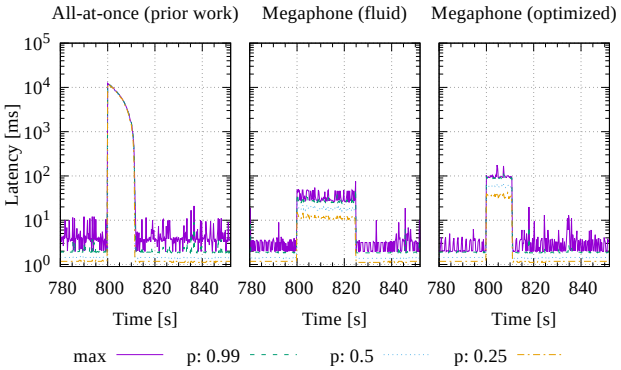


Figure 1: A comparison of service latencies in prior coarse-grained migration strategies (all-at-once) with two of Megaphone's fine-grained migration strategies (fluid and optimized), for a workload that migrates one billion keys consisting of 8 GiB of data.

State migration has been extensively studied for distributed databases [8, 11, 13, 12]. Notably, Squall [12] uses transactions to multiplex fine-grained state migration with data processing. These techniques are appealing in spirit, but use mechanisms (transactions, locking) not available in high-throughput stream processors and are not directly applicable without significant performance degradation.

In this paper we present Megaphone, a technique for fine-grained migration in a stream processor which delivers maximum latencies orders of magnitude lower than existing techniques, based on the observation that a stream processor's structured computation and logical timestamps allow the system to *plan* fine-grained migrations. Megaphone can specify migrations on a key-by-key basis, and then optimizes this by batching at varying granularities; as Figure 1 shows, the improvement over all-at-once migration can be dramatic. This paper is an extended version of a preliminary workshop publication [19]. In this paper, we describe a more general mechanism, further detail its implementation, and evaluate it more thoroughly on realistic workloads.

Our main contribution is *fluid migration* for stateful streaming dataflows: a state migration technique that enables consistent online reconfiguration of streaming dataflows and smoothens latency spikes without using additional resources (Section 3) by employing fine-grained planning and coordination through logical timestamps. Additionally, we design and implement an API for reconfigurable stateful timely dataflow operators that enables fluid migration to be

Proceedings of the VLDB Endowment, Vol. 12, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3329772.3329777

controlled simply through additional dataflow streams rather than through changes to the dataflow runtime itself (Section 4). Finally, we show that Megaphone has negligible steady-state overhead and enables fast direct state movement using the NEXMark benchmarks suite and microbenchmarks (Section 5).

Megaphone is built on timely dataflow,[1] and is implemented purely in library code, requiring no modifications to the underlying system. We first review existing state migration techniques in streaming systems, which either cause performance degradation or require resource overprovisioning. We also review live migration in DBMSs and identify the technical challenges to implement similar methods in distributed stream processors (Section 2).

2 Background and Motivation

A distributed dataflow computation runs as a physical execution plan which maps operators to provisioned compute resources (or workers). The execution plan is a directed graph whose vertices are operator instances (each on a specific worker) and edges are data channels (within and across workers). Operators can be stateless (e.g., filter, map) or stateful (e.g., windows, rolling aggregates). State is commonly partitioned by key across operator instances so that computations can be executed in a data-parallel manner. At each point in time of a computation, each worker (with its associated operator instances) is responsible for a set of keys and their associated state.

State migration is the process of changing the assignment of keys to workers and redistributing respective state accordingly. A good state migration technique should be *non-disruptive* (minimal increase in response latency during migration), *short-lived* (migration completes within a short period of time), and *resource-efficient* (minimal additional resources required during the migration).

We present an overview of existing state migration strategies in distributed streaming systems and identify their limitations. We then review live state migration methods adopted by database systems and provide an intuition into Megaphone's approach to bring such migration techniques to streaming dataflows.

2.1 State migration in streaming systems

Distributed stream processors, including research prototypes and production-ready systems, use one of the following three state migration strategies.

Stop-and-restart A straight-forward way to realize state migration is to temporarily stop program execution, safely transfer state when no computation is being performed, and restart the job once state redistribution is complete. This approach is most commonly enabled by leveraging existing fault-tolerance mechanisms in the system, such as global state checkpoints. It is adopted by Spark Streaming [29], Structured Streaming [7], and Apache Flink [9].

Partial pause-and-resume In many reconfiguration scenarios only one or a small number of operators need to migrate state, and halting the entire dataflow is usually unnecessary. An optimization first introduced in Flux [26] and later adopted in variations by Seep [15], IBM Streams [2], StreamCloud [17], Chi [21], and FUGU [18], pauses the computation only for the affected dataflow subgraph. Operators not participating in the migration continue without interruption. This approach can use fault-tolerance checkpoints for state migration as in [15, 21] or state can be directly migrated between operators as in [17, 18].

[1] `https://github.com/frankmcsherry/timely-dataflow`

Dataflow Replication To minimize performance penalties, some systems replicate the whole dataflow or subgraphs of it and execute the old and new configurations in parallel until migration is complete. ChronoStream [28] concurrently executes two or more computation slices and can migrate an arbitrary set of keys between instances of a single dataflow operator. Gloss [24] follows a similar approach and gathers operator state during a migration in a centralized controller using an asynchronous protocol.

Current systems fall short of implementing state migration in a non-disruptive and cost-efficient manner. Existing stream processors migrate state *all-at-once*, but differ in whether they pause the existing computation or start a concurrent computation. As Figure 1 shows, strategies that pause the computation can cause high latency spikes, especially when the state to be moved is large. On the other hand, dataflow replication techniques reduce the interruption, but at the cost of high resource requirements and required support for input duplication and output de-duplication. For example, for ChronoStream to move from a configuration with x instances to a new one with y instances, $x + y$ instances are required during the migration.

2.2 Live migration in database systems

Database systems have implemented optimizations that explicitly target limitations we have identified in the previous section, namely unavailability and resource requirements. Even though streaming dataflow systems differ significantly from databases in terms of data organization, workload characteristics, latency requirements, and runtime execution, the fundamental challenges of state migration are common in both setups.

Albatross [11] adopts VM live migration techniques and is further optimized in [8] with a dynamic throttling mechanism, which adapts the data transfer rate during migration so that tenants in the source node can always meet their SLOs. Prorea [25] combines push-based migration of hot pages with pull-based migration of cold pages. Zephyr [13] proposes a technique for live migration in shared-nothing transactional databases which introduces no system downtime and does not disrupt service for non-migrating tenants.

The most sophisticated approach is Squall [12], which interleaves state migration with transaction processing by (in part) using transaction mechanisms to effect the migration. Squall introduces a number of interesting optimizations, such as pre-fetching and splitting reconfigurations to avoid contention on a single partition. In the course of a migration, if migrating records are needed for processing but not yet available, Squall introduces a transaction to acquire the records (completing their migration). This introduces latency along the critical path, and the transaction locking mechanisms can impede throughput, but the system is neither paused nor replicated. To the best of our knowledge, no stream processor implements such a fine-grained migration technique.

2.3 Live migration for streaming dataflows

Applying existing fine-grained live migration techniques to a streaming engine is non-trivial. While systems like Squall target OLTP workloads with short-lived transactions, streaming jobs are long-running. In such a setting, Squall's approach to acquire a global lock during initialization is not a viable solution. Further, many of Squall's remedies are reactive rather than proactive (because it must support general transactions whose data needs are hard to anticipate), which can introduce significant latency on the critical path.

The core idea behind Megaphone's migration mechanism is to multiplex *fine-grained* state migration with data processing, coordinated using logical time common in stream processors. This is a proactive approach to migration that relies on the prescribed structure of streaming computations, and the ability of stream processors

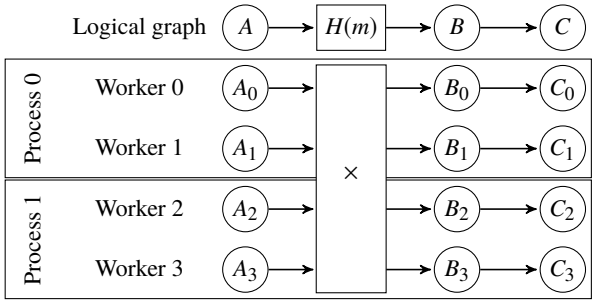


Figure 2: Timely dataflow execution model

to coordinate with high frequency using logical time. Such systems, including Megaphone, avoid the need for system-wide locks by pre-planning the rendezvous of data at specific workers.

3 State migration design

Megaphone's features rely on core streaming dataflow concepts such as logical time, progress tracking, data-parallel operators, and state management. Basic implementations of these concepts are present in all modern stream processors, such as Apache Flink [10], Millwheel [5], and Google Dataflow [6]. In the following, we rely on the Naiad [22] timely dataflow model as the basis to describe the Megaphone migration mechanism. Timely dataflow natively supports a superset of dataflow features found in other systems in their most general form.

3.1 Timely dataflow concepts

A streaming computation in Naiad is expressed as a *timely dataflow*: a directed (possibly cyclic) graph where nodes represent stateful operators and edges represent data streams between operators. Each data record in timely dataflow bears a *logical timestamp*, and operators maintain or possibly advance the timestamps of each record. Example timestamps include integers representing milliseconds or transaction identifiers, but in general can be any set of opaque values for which a partial order is defined. The timely dataflow system tracks the existence of timestamps, and reports as processed timestamps no longer exist in the dataflow, which indicates the forward progress of a streaming computation.

A timely dataflow is executed by multiple *workers* (threads) belonging to one or more OS processes, which may reside in one or more machines of a networked cluster. Workers communicate with each other by exchanging messages over data channels (shared-nothing paradigm) as shown in Figure 2. Each worker has a local copy of the entire timely dataflow graph and executes all operators in this graph on (disjoint) partitions of the dataflow's input data. Each worker repeatedly executes dataflow operators concurrent with other workers, sending and receiving data across data exchange channels. Due to this asynchronous execution model, the presence of concurrent "in-flight" timestamps is the rule rather than the exception.

As timely workers execute, they communicate the numbers of logical timestamps they produce and consume to all other workers. This information allows each worker to determine the possibility that any dataflow operator may yet see any given timestamp in its input. The timely workers present this information to operators in the form of a *frontier*:

Definition 1. A **frontier** F is a set of logical timestamps such that
1. no element of F is strictly greater than another element of F,
2. all timestamps on messages that may still be received are greater than or equal to some element of F.

In many simple settings a frontier is analogous to a *low watermark* in streaming systems like Flink, which indicates the single smallest timestamp that may still be received. In timely dataflow a frontier must be set-valued rather than a single timestamp because timestamps may be only partially ordered.

Operators in timely dataflow may retain *capabilities* that allow the operator to produce output records with a given timestamp. All received messages come bearing such a capability for their timestamp. Each operator can choose to drop capabilities, or downgrade them to later timestamps. The timely dataflow system tracks capabilities held by operators, and only advances downstream frontiers as these capabilities advance.

Timely dataflow frontiers are the main mechanism for coordination between otherwise asynchronous workers. The frontiers indicate when we can be certain that all messages of a certain timestamp have been received, and it is now safe to take any action that needed to await their arrival. Importantly, frontier information is entirely passive and does not interrupt the system execution; it is up to operators to observe the frontier and determine if there is some work that cannot yet be performed. This enables very fine-grained coordination, without system-level intervention. Further technical details of progress tracking in timely dataflows can be found in [22, 4].

We will use timely dataflow frontiers to separate migrations into independent arbitrarily fine-grained timestamps and logically coordinate data movement without using coarse-grained pause-and-resume for parts of the dataflow.

3.2 Migration formalism and guarantees

To frame the mechanism we introduce for live migration in streaming dataflows, we first lay out some formal properties that define correct and live migration. In the interest of clarity we keep the descriptions casual, but each can be formalized.

We consider stateful dataflow operators that are *data-parallel* and *functional*. Specifically, an operator acts on input data that are structured as (key, val) pairs, each bearing a logical timestamp. The input is partitioned by its *key* and the operator acts independently on each input partition by sequentially applying each *val* to its state in timestamp order. For each *key*, for each *val* in timestamp order, the operator may change its per-key state arbitrarily, produce arbitrary outputs as a result, and it may schedule further per-key changes at future timestamps (in effect sending itself a new, post-dated *val* for this *key*).

$$operator_{key}\colon (state, val) \rightarrow (state', [outputs], [(vals, times)])$$

The output triples are the new state, the outputs to produce, and future changes that should be presented to the operator.

For a specific *operator*, we can describe the correctness of an implementation. We introduce the notation of *in advance of* as follows.

Definition 2 (in advance of)**.** A timestamp t is **in advance of**
1. a timestamp t' if t is greater than or equal to t';
2. a frontier F if t is greater than or equal to an element of F.

In-advance-of corresponds to the less-or-equal relation for partially ordered sets. For example, a time 6 is in advance of 5.

Property 1 (Correctness)**.** The **correct outputs through *time*** are the timestamped outputs that result from each key from the timestamp-ordered application of input and post-dated records bearing timestamp not in advance of *time*.

For each migrateable operator, we also consider a *configuration* function, which for each timestamp assigns each key to a specific worker.

$$\textit{configuration}\colon (\textit{time}, \textit{key}) \rightarrow \textit{worker}$$

For example, the configuration function could assign a key `a` to worker 2 for times $[4, 8)$ and to worker 1 for times $[8, 16)$.

With a specific *configuration*, we can describe the correctness of a migrating implementation.

Property 2 (Migration)**.** A computation is **migrated according to *configuration*** if all updates to *key* with timestamp *time* are performed at worker *configuration(time, key)*.

A configuration function can be represented in many ways, which we will discuss further. In our context we will communicate any changes using a timely dataflow stream, in which configuration changes bear the logical timestamp of their migration. This choice allows us to use timely dataflow's frontier mechanisms to coordinate migrations, and to characterize liveness.

Property 3 (Completion (liveness))**.** A migrating computation is **completing** if, once the frontiers of both the data input stream and configuration update stream reach F, then (with no further requirements of the input) the output frontier of the computation will eventually reach F.

Our goal is to produce a mechanism that satisfies each of these three properties: Correctness, Migration, and Completion.

3.3 Configuration updates

State migration is driven by updates to the *configuration* function introduced in 3.2. In Megaphone these updates are supplied as data along a timely dataflow stream, each bearing the logical timestamp at which they should take effect. Informally, configuration updates have the form

$$\textit{update}\colon (\textit{time}, \textit{key}, \textit{worker})$$

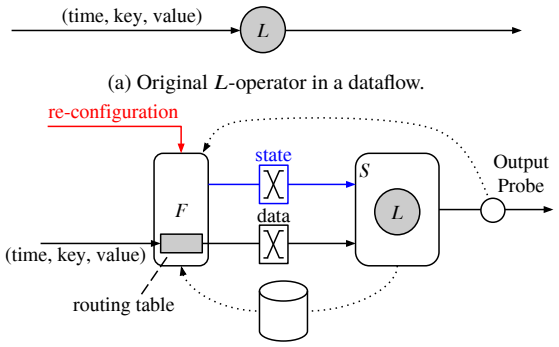
indicating that as of *time* the state and values associated with *key* will be located at *worker*, and that this will hold until a new update to *key* is observed with a greater timestamp. For example, an update could have the form of (*time:* 16, *key:* `a`, *worker:* 0), which would define the configuration function for times of 16 and beyond.

As configuration updates are simply data, the user has the ability to drive a migration process by introducing updates as they see fit. In particular, they have the flexibility to break down a large migration into a sequence of smaller migrations, each of which have lower duration and between which the system can process data records. For example, to migrate from one configuration C_1 to another C_2, a user can use different migration strategies to reveal the changes from C_1 to C_2:

All-at-once migration To simultaneously migrate all keys from C_1 to C_2, a user could supply all changed (*time*, *key*, *worker*) triples with one common *time*. This is essentially an implementation of the *partial pause-and-restart* migration strategy of existing streaming systems as described in Section 2.1.

Fluid migration To smoothly migrate keys from C_1 to C_2, a user could repeatedly choose one key changed from C_1 to C_2, introduce the new (*time*, *key*, *worker*) triple with the current *time*, and await the migration's completion before choosing the next key.

Batched migration To trade off low latency against high throughput, a user can produce batches of changed (*time*, *key*, *worker*) triples with a common *time*, awaiting the completion of the batch before introducing the next batch of changes.



(a) Original L-operator in a dataflow.

(b) Megaphone's operator structure in a dataflow.

Figure 3: Overview of Megaphone's migration mechanism

We believe that this approach to reconfiguration, as user-supplied data, opens a substantial design space. Not only can users perform fine-grained migration, they can prepare future migrations at specific times, and drive migrations based on timely dataflow computations applied to system measurements. Most users will certainly need assistance in performing effective migration, and we will evaluate several specific instances of the above strategies.

3.4 Megaphone's mechanism

We now describe how to create a migrateable version of an operator L implementing some deterministic, data-parallel *operator* as described in 3.2. A non-migrateable implementation would have a single dataflow operator with a single input dataflow stream of (*key*, *val*) pairs, exchanged by *key* before they arrive at the operator.

Instead, we create two operators F and S. F takes the data stream and the stream of configuration updates as an additional input and produces data pairs and migrating state as outputs. The configuration stream can be ingested from an external controller such as DS2 [20] or Chi [21]. S takes as inputs exchanged data pairs and exchanged migrating state, and applies them to a hosted instance of L, which implements *operator* and maintains both state and pending records for each key. Figure 3b presents a schematic overview of the construction. Recall that in timely dataflow instances of all operators in the dataflow are multiplexed on each worker (core). The F and S on the same worker share access to L's state.

This construction can be repeated for all the operators in the dataflow that need support for migration. Separate operators can be migrated independently (via separate configuration update streams), or in a coordinated manner by re-using the same configuration update stream. Operators with multiple data inputs can be treated like single-input operators where the migration mechanism acts on both data inputs at the same time.

Operator F Operator F routes (*key*, *val*) pairs according to the configuration at their associated *time*, buffering pairs if *time* is in advance of the frontier of the configuration input. For times in advance of this frontier, the configuration is not yet certain as further configuration updates could still arrive. The configuration at times not in advance of this frontier can no longer be updated. As the data frontier advances, configurations can be retired.

Operator F is also responsible for initiating state migrations. For a configuration update (*time*, *key*, *worker*), F must not initiate a migration for *key* until its state has absorbed all updates at times strictly less than *time*. F initiates a migration once *time* is present in the output frontier of S, as this implies that there exist no records at timestamps less than *time*, as otherwise they would be present in the frontier in place of *time*.

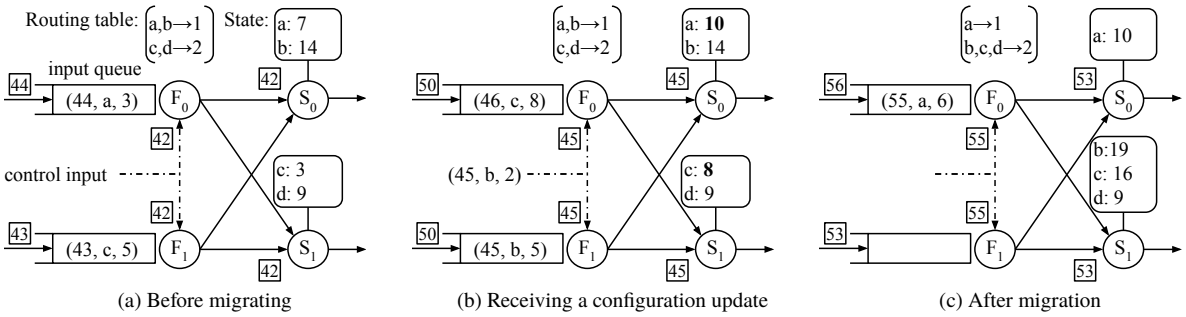


Figure 4: A migrating word-count dataflow executed by two workers. The example is explained in more detail in Section 3.5

Operator F initiates a migration by uninstalling the current state for *key* from its current location in operator S, and transmitting it bearing timestamp *time* to the instance of operator S on *worker*. The state includes both the state for *operator*, as well as the list of pending (*val*, *time*) records produced by *operator* for future times.

Operator S Operator S receives exchanged (*key*, *val*) pairs and exchanged state as the result of migrations initiated by F. S immediately installs any received state. S applies received and pending (*key*, *val*) pairs in timestamp order using *operator* once their timestamp is not in advance of either the data or state inputs.

We provide details of Megaphone's implementation of this mechanism in Section 4.

Proof sketch For each key, $operator_{key}$ defines a timeline corresponding to a single-threaded execution, which assigns to each time a pair (*state*, [(*val*, *time*)]) of state and pending records *just before* the application of input records at that time. Let $P(t)$ denote the function from times to these pairs for *key*.

For each key, the *configuration* function partitions this timeline into disjoint intervals, $[t_a, t_b)$, each of which is assigned to one operator instance S_a.

Claim: F migrates exactly $P(t_a)$ to S_a.

First, F always routes input records at *time* to S_a, and so routes all input records in $[t_a, t_b)$ to S_a. If F also presents S_a with $P(t_a)$, it has sufficient input to produce $P(t_b)$. More precisely,

1. Because F maintains its output frontier at t_b, in anticipation of the need to migrate $P(t_b)$, S_a will apply no input records in advance of t_b. And so, it applies exactly the records in $[t_a, t_b)$.
2. Until S_a transitions to $P(t_b)$, its output frontier will be strictly less than t_b, and so F will not migrate anything other than $P(t_b)$.
3. Because F maintains its output frontier at t_b, and S_a is able to advance its output frontier to t_b, the time t_b will eventually be *in* the output frontier of S.

3.5 Example

Figure 4 presents three snapshots of a migrating streaming word-count dataflow. The figure depicts operator instances F_0 and F_1 of the upstream routing operator, and operator instances S_0 and S_1 of the operator instances hosting the word-count state and update logic. The F operators maintain input queues of received but not yet routable input data, and an input stream of logically timestamped configuration updates. Although each F maintains its own routing table, which may temporarily differ from others, we present one for clarity. Input frontiers are represented by boxed numbers, and indicate timestamps that may still arrive on that input.

In Figure 4a, F_0 has enqueued the record (44, `a`, 3) and F_1 has enqueued the record (43, `c`, 5), both because their control input frontier has only reached 42 and so the destination workers at their associated timestamps have not yet been determined. Generally, F instances will only enqueue records with timestamps in advance of the control input frontier, and the output frontiers of the S instances can reach the minimum of the data and control input frontiers.

In Figure 4b, both control inputs have progressed to 45. The buffered records (44, `a`, 3) and (43, `c`, 5) have been forwarded to S_1 and S_2, and the count operator instances apply the state updates accordingly, shown in bold. Additionally, both operators have received a configuration update for the key `b` at time 45. Should the configuration input frontier advance beyond 45, both F_0 and F_1 can integrate the configuration change, and then react. Operator F_0 would observe that the output frontier of S_0 reaches 45, and initiate a state migration. Operator F_1 would route its buffered input at time 45, to S_1 rather than S_0.

In Figure 4c the migration has completed. Although the configuration frontier has advanced to 55, the output frontiers are held back by the data input frontier of F_1 at 53. According to Definition 1, the frontier guarantees that no record with a time earlier than 53 will appear at the input. If the configuration frontier advances past 55 then operator F_0 could route its queued record, but neither S operator could apply it until they are certain that there are no other data records that could come before the record at 55.

4 Implementation

Megaphone is an implementation of the migration mechanism described in Section 3. In this section, we detail specific choices made in Megaphone's implementation, including the interfaces used by the application programmer, Megaphone's specific choices for the grouping and organization of per-key state, and how we implemented Megaphone's operators in timely dataflow. We conclude with some discussion of how one might implement Megaphone in other stream processing systems, as well as alternate implementation choices one could consider.

4.1 Megaphone's operator interface

Megaphone presents users with an operator interface that closely resembles the operator interfaces timely dataflow presents. In several cases, users can use the same operator interface extended only with an additional input stream for configuration updates. More generally, we introduce a new structure to help users isolate and surface all information that must be migrated (state, but also pending future records). These additions are implemented strictly above timely dataflow, but their structure is helpful and they may have value in timely dataflow proper.

The simplest stateful operator interface Megaphone and timely provide is the `state_machine` operator, which takes one input structured as pairs (*key*, *val*) and a state update function which can

```
fn state_machine(
  control: Stream<ControlInstr>,
  input: Stream<(K, V)>,
  exchange: K -> Integer
  fold: |Key, Val, State| -> List<Output>
) -> Stream<Output>;

fn unary(
  control: Stream<ControlInstr>,
  input: Stream<Data>,
  exchange: Data -> Integer,
  fold: |Time, Data, State, Notificator| -> List<Output>
) -> Stream<Output>;

fn binary(
  control: Stream<ControlInstr>,
  input1: Stream<Data1>, input2: Stream<Data2>,
  exchange1: Data1 -> Integer,
  exchange2: Data2 -> Integer,
  fold: |Time, Data1, Data2, State, Notificator1,
       Notificator2| -> List<Output>
) -> Stream<Output>;
```

Listing 1: Abstract definition of the Megaphone operator interfaces. Arguments `State` and `Notificator` are provided as mutable references which can be operated upon.

```
worker.dataflow(|scope| {
  // Introduce configuration and input streams.
  let conf = conf_input.to_stream(scope);
  let text = text_input.to_stream(scope);

  // Update per-word accumulate counts.
  let count_stream = megaphone::unary(
    conf,
    text,
    |(word, diff)| hash(word),
    |time, data, state, notificator| {
      // map each (word, diff) pair to the accumulation.
      data.map(|(word, diff)| {
        let mut count = state.entry(word).or_insert(0);
        *count += diff;
        (word, count)
      })
    }
  );
});
```

Listing 2: A stateful word-count operator. The operator reads (*word*, *diff*)-pairs and outputs the accumulated count of each encountered word. For clarity, the example suppresses details related to Rust's data ownership model.

produce arbitrary output as it changes per-key state in response to keys and values. In Megaphone, there is an additional input for configuration updates, but the operator signature is otherwise identical.

More generally, timely dataflow supports operators of arbitrary numbers and types of inputs, containing arbitrary user logic, and maintaining arbitrary state. In each case a user must specify a function from input records to integer keys, and the only guarantee timely dataflow provides is that records with the same key are routed to the same worker. Operator execution and state are partitioned by worker, but not necessarily by key.

For Megaphone to isolate and migrate state and pending work we must encourage users to yield some of the generality timely dataflow provides. However, timely dataflow has already required the user to program partitioned operators, each capable of hosting multiple keys, and we can lean on these idioms to instantiate more fine-grained operators, partitioned not only by worker but further into finer-grained *bins* of keys. Routing functions for each input are already required by timely dataflow, and Megaphone interposes to allow the function to change according to reconfiguration. Timely dataflow per-worker state is defined implicitly by the state captured by the operator closure, and Megaphone only makes it more explicit. The use of a helper to enqueue pending work is borrowed from an existing timely dataflow idiom (the `Notificator`). While Megaphone's general API is not identical to that of timely dataflow, it is just a more crisp framing of the same idioms.

Listing 1 shows how Megaphone's operator interface is structured. The interface declares unary and binary stateful operators for single input or dual input operators as well as a state-machine operator. The logic for the state-machine operator has to be encoded in the `fold`-function. Megaphone presents data in timestamp order with a corresponding state and notificator object. Here, migration is transparent and performed without special handling by the operator implementation.

Example Listing 2 shows an example of a stateful word-count dataflow with a single data input and an additional `control` input. The `stateful_unary` operator receives the `control` input, the state type, and a key extraction function as parameters. The control input carries information about where data is to be routed as discussed in the previous section. During migration, the state object is converted into a stream of serialized tuples, which are used to reconstruct the object on the receiving worker. State is managed in groups of keys, i.e. many keys of input data will be mapped to the same state object. The key extraction function defines how this key can be extracted from the input records.

4.2 State organization

State migration as defined in Section 3.2 is defined on a per-key granularity. In a typical streaming dataflow, the number of keys can be large in the order of million or billions of keys. Managing each key individually can be costly and thus we selected to group keys into *bins* and adapt the *configuration* function as follows:

$$\mathit{configuration} : (\mathit{time}, \mathit{bin}) \rightarrow \mathit{worker}.$$

Additionally, each key is statically assigned to one equivalence class that identifies the bin it belongs to.

In Megaphone, the number of bins is configurable in powers of two at startup but cannot be changed during run-time. A stateful operator gets to see a bin that holds data for the equivalence class of keys for the current input. Bins are simply identified by a number, wich corresponds to the most significant bits of the exchange function specified on the operator.[2]

Megaphone's mechanism requires two distinct operators, F and S. The operator S maintains the bins local to a worker and passes references to the user logic L. Nevertheless, the S-operator does not have a direct channel to its peers. For this reason, F can obtain a reference to bins by means of a shared pointer. During a migration, F serializes the state obtained via the shared pointer and sends it to the new owning S-operator via a regular timely dataflow channel. Note that sharing a pointer between two operators requires the operators to be executed by the same process (or thread to avoid synchronization), which is the case for timely dataflow.

[2]Otherwise, keys with similar least-significant bits are mapped to the same bin; Rust's `HashMap`-implementation suffers from collisions for keys with similar least-significant bits.

4.3 Timely instantiation

In timely dataflow, data is exchanged according to an *exchange* function, which takes some data and computes an integer representation:

$$exchange : data \rightarrow Integer.$$

Timely dataflow uses this value to decide where to send tuples. In Megaphone, we introduce an indirection layer where bins are assigned to workers. That way, the exchange function for the channels from F to S is by a specific worker identifier.

Monitoring output frontiers Megaphone ties migrations to logical time and a computation's progress. A reconfiguration at a specific time is only to be applied once all data up to that time has been processed. The F operators access this information by monitoring the output frontier of the S operators. Specifically, timely dataflow supports *probes* as a mechanism to observe progress on arbitrary dataflow edges. Each worker attaches a probe to the output stream of the S operators, and provides the probe to its F operator instance.

Capturing timely idioms For Megaphone to migrate state, it requires clear isolation of per-key state and pending records. Although timely dataflow operators require users to write operators that can be partitioned across workers, they do not require the state and pending records to be explicitly identified. To simplify programming migrateable operators, we encapsulate several timely dataflow idioms in a helper structure that both manages state and pending records for the user, and surfaces them for migration.

Timely dataflow has a `Notificator` type that allows an operator to indicate future times at which the operator may produce output, but without encapsulating the keys, states, or records it might use. We implemented an extended notificator that buffers future triples (*time*, *key*, *val*) and can replay subsets for times not in advance of an input frontier. Internally the triples are managed in a priority queue, unlike in timely dataflow, which allows Megaphone to efficiently maintain large numbers of future triples. By associating data (keys, values) with the times, we relieve the user from maintaining this information on the side. As we will see, Megaphone's notificator can result in a net reduction in implementation complexity, despite eliciting more information from the user.

4.4 Discussion

Up to now, we explained how to map the abstract model of Megaphone to an implementation. The model leaves many details to the implementation, several of which have a large effect on an implementation's run-time performance. Here, we want to point out how they interact with other features of the underlying system, what possible alternatives are and how to integrate Megaphone into a larger, controller-based system.

Other systems We implemented Megaphone in timely dataflow, but the mechanisms could be applied on any sufficiently expressive stream processor with support for event time, progress tracking, and state management. Specifically, Megaphone relies on the ability of F operators to 1. observe timestamp progress at other locations in the dataflow, and 2. to extract state from downstream S operators for migration. With regard to first requirement, systems with out-of-band progress tracking like Millwheel [5] and Google Dataflow [6] also provide the capability to observe dataflow progress externally, while systems with in-band watermaks like Flink would need to provide an additional mechanism. Extracting state from downstream operators is straight-forward in timely dataflow where workers manage multiple operators. In systems where each thread of control manages a single operator external coordination and communication mechanisms could be used to effect the same behavior.

Fault tolerance Megaphone is a library built on timely dataflow abstractions, and inherits fault-tolerance guarantees from the system. For example, the Naiad implementation of timely dataflow provides system-wide consistent snapshots, and a Megaphone implementation on Naiad would inherit fault tolerance. At the same time, Megaphone's migration mechanisms effectively provide programmable snapshots on finer granularities, which could feed back into finer-grained fault-tolerance mechanisms.

Alternatives to binning Megaphone's implementation uses binning to reduce the complexity of the *configuration* function. An alternative to a static mapping of keys to bins could be achieved by the means of a prefix tree (e.g., a longest-prefix match as in Internet routing tables). Runtime-reconfiguration of the binning strategy could be enabled by splitting and merging bins.

Migration controller We implemented Megaphone as a system that provides an input for configuration updates to be supplied by an external controller. The only requirement Megaphone places on the controller is to adhere to the control command format as described in Section 3.3. A controller could observe the performance characteristics of a computation on a per-key level and correlate this with the input workload. For example, the recent DS2 [20] system automatically measures and re-scales streaming systems to meet throughput targets. Megaphone can also be driven by general re-configuration controllers and is not restricted to elasticity policies. For instance, the configuration stream could be provided by Dhalion [16] or Chi [21].

Independently, we have observed and implemented several details for effective migration. Specifically, we can use bipartite matching to group migrations that do not interfere with each other, reducing the number of migration steps without much increasing the maximum latency. We can also insert a gap between migrations to allow the system to immediately drain enqueued records, rather than during the next migration, which reduces the maximum latency from two migration durations to just one.

5 Evaluation

Our evaluation of Megaphone is in three parts. We are interested in particular in the latency of streaming queries, and how they are affected by Megaphone both in a steady state (where no migration is occuring) and during a migration operation.

First, in Section 5.1 we use the NEXMark benchmarking suite [23, 27] to compare Megaphone with prior techniques under a realistic workload. Next, in Section 5.2 we look at the overhead of Megaphone when no migration occurs: this is the cost of providing migration functionality in stateful dataflow operators, versus using optimized operators which cannot migrate state. Finally, in Section 5.3 we use a microbenchmark to investigate how parameters like the number of bins and size of the state affect migration performance.

We run all experiments on a cluster of four machines, each with four Intel Xeon E5-4650 v2 @2.40 GHz CPUs (each 10 cores with hyperthreading) and 512 GiB of RAM, running Ubuntu 18.04. For each experiment, we pin a timely process with four workers to a single CPU socket. Our open-loop testing harness supplies the input at a specified rate, even if the system itself becomes less responsive (e.g., during a migration). We record the observed latency every 250 ms, in units of nanoseconds, which are recorded in a histogram of logarithmically-sized bins.

Unless otherwise specified, we migrate the state of the main operator of each dataflow. We first migrate half of the keys on half of the workers to the other half of the workers (25% of the total state), resulting in an imbalanced assignment. We then perform and report the details of a second migration back to the balanced configuration.

Table 1: NEXMark query implementations lines of code.

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Native	12	14	58	128	73	130	55	58
Megaphone	16	18	41	74	46	74	54	29

5.1 NEXMark Benchmark

The NEXMark suite models an auction site in which a high-volume stream of users, auctions, and bids arrive, and eight standing queries are maintained reflecting a variety of relational queries including stateless streaming transformations (e.g., map and filter in Q1 and Q2 respectively), a stateful record-at-a-time two-input operator (incremental join in Q3), and various window operators (e.g., sliding window in Q5, tumbling window join in Q8), and complex multi-operator dataflows with shared components (Q4 and Q6).

We have implemented all eight of the NEXMark queries in both native timely dataflow and using Megaphone. Table 1 lists the lines of code for queries 1–8. *Native* is a hand-tuned implementation, *Megaphone* is implemented using the stateful operator interface. Note that the implementation complexity for the native implementation is higher in most cases as we include optimizations from Section 4 which are not offered by the system but need to be implemented for each operator by hand.

To test our hypothesis that Megaphone supports efficient migration on realistic workloads, we run each NEXMark query under high load and migrate the state of each query without interrupting the query processing itself. Our test harness uses a reference input data generator and increases its rate. The data generator can be played at a higher rate but this does not change certain intrinsic properties. For example, the number of active auctions is static, and so increasing the event rate decreases auction duration. For this reason, we present time-dilated variants of queries Q5 and Q8 containing large time-based windows (up to 12 hours). We run all queries with 4×10^6 updates per second. For stateful queries, we perform a first migration at 400 s and perform and report a second re-balancing migration at 800 s. We compare *all-at-once*, which is essentially equivalent to the partial pause-and-restart strategy adopted by existing systems, and *batched*, Megaphone's optimized migration strategy which strikes a balance between migration latency and duration (cf. Section 3.3). We use 2^{12} bins for Megaphone's migration; in Section 5.2 we study Megaphone's sensitivity to the bin count.

Figures 7 through 12 show timelines for the second migration of stateful queries Q3 through Q8. Generally, the all-at-once migrations experience maximum latencies proportional to the amount of state maintained, whereas the latencies of Megaphone's batched migration are substantially lower when the amount of state is large.

Query 1 and Query 2 maintain no state. Q1 transforms the stream of bids to use a different currency, while Q2 filters bids by their auction identifiers. Despite the fact that both queries do not accumulate state to migrate, we demonstrate their behavior to establish a baseline for Megaphone and our test harness. Figures 5 and 6 show query latency during two migrations where no state is thus transferred; any impact is dominated by system noise.

Query 3 joins auctions and people to recommend local auctions to individuals. The join operator maintains the auctions and people relations, using the seller and person as the keys, respectively. This state grows without bound as the computation runs. Figure 7 shows the query latency for both Megaphone, and the native timely implementation. We note that while the native timely implementation has some spikes, they are more pronounced in Megaphone, whose tail latency we investigate further in Section 5.2.

Query 4 reports the average closing prices of auctions in a category relying on a stream of closed auctions, derived from the streams

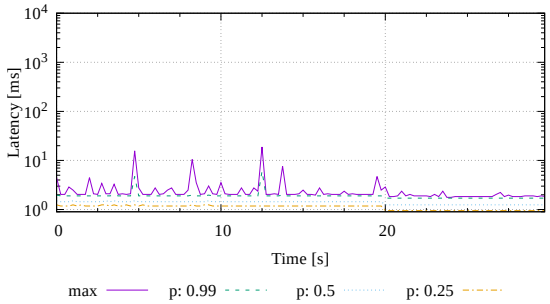


Figure 5: NEXMark query latency for Q1, 4×10^6 requests per second, reconfiguration at 10 s and 20 s. No latency spike occurs during migration as the query does not accumulate state.

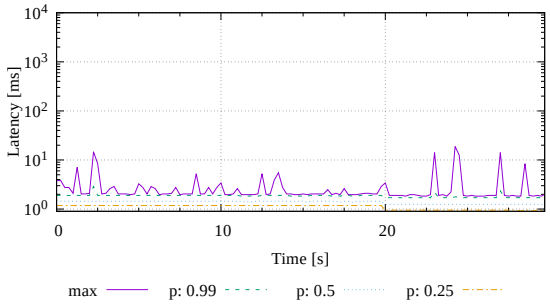
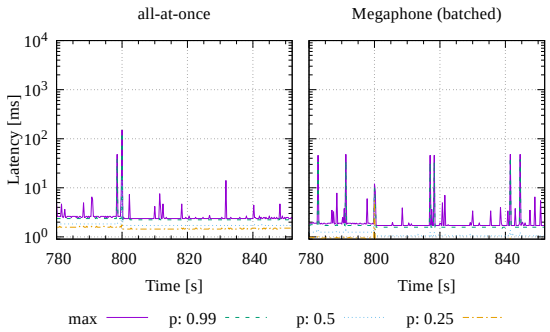
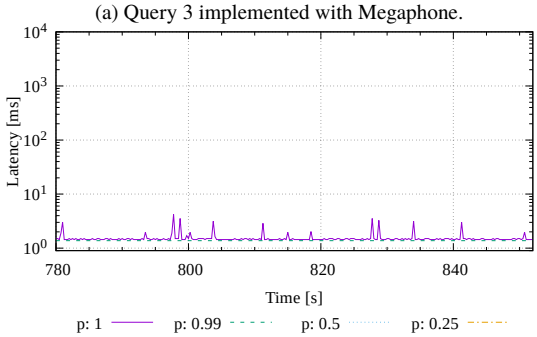


Figure 6: NEXMark query latency for Q2, 4×10^6 requests per second, reconfiguration at 10 s and 20 s. No latency spike occurs during migration as the query does not accumulate state.



(a) Query 3 implemented with Megaphone.



(b) Query 3 native implementation.

Figure 7: NEXMark query latency for Q3. A small latency spike can be observed at 800 s for both all-at-once and batched migration strategies, reaching more than 100 ms for all-at-once and 10 ms for batched migration. Although the state for query 3 grows without bounds, this did not bear significance after 800 s.

of bids and auctions, which we compute and maintain, and contains one operator keyed by auction id which accumulates relevant bids until the auction closes, at which point the auction is reported and removed. The NEXMark generator is designed to have a fixed number of auctions at a time, and so the state remains bounded. Figure 8 shows the latency timeline during the second migration. The all-at-once migration strategy causes a latency spike of more than two seconds whereas the batched migration strategy only shows an increase in latency of up to 100 ms.

Query 5 reports, each minute, the auctions with the highest number of bids taken over the previous sixty minutes. It maintains up to sixty counts for each auction, so that it can both report and retract counts as time advances. To elicit more regular behavior, our implementation reports every second over the previous minute, effectively dilating time by a factor of 60. Figure 9 shows the latency timeline for the second migration; the all-at-once migration is an order of magnitude larger than the per-second events, whereas Megaphone's batched migration is not distinguishable.

Query 6 reports the average closing price for the last ten auctions of each seller. This operator is keyed by auction seller, and maintains a list of up to ten prices. As the computation proceeds, the set of sellers, and so the associated state, grows without bound. Figure 10 shows the timeline at the second migration. The result is similar to query 4 because both share a large fraction of the query plan.

Query 7 reports the highest bid each minute, and the results are shown in Figure 11. This query has minimal state (one value) but does require a data exchange to collect worker-local aggregations to produce a computation-wide aggregate. Because the state is so small, there is no distinction between all-at-once and batched migration.

Query 8 reports a twelve-hour windowed join between new people and new auction sellers. This query has the potential to maintain a massive amount of state, as twelve hours of auction and people data is substantial. Once reached, the peak size of state is maintained. To show the effect of twelve-hour windows, we dilate the internal time by a factor of 79. The reconfiguration time of 800 s corresponds to approximately 17.5 h of event time.

These results show that for NEXMark queries maintaining large amounts of state, all-at-once migration can introduce significant disruption, which Megaphone's batched migration can mitigate. In principle, the latency could be reduced still further with the fluid migration strategy, which we evaluate in Section 5.3.

5.2 Overhead of the interface

We now use a counting microbenchmark to measure the overhead of Megaphone, from which one can determine an appropriate trade-off between migration granularity and this overhead. We compare Megaphone to native timely dataflow implementations, as we vary the number of bins that Megaphone uses for state. We anticipate that this overhead will increase with the number of bins, as Megaphone must consult a larger routing table.

The workload uses a stream of randomly selected 64-bit integer identifiers, drawn uniformly from a domain defined per experiment. The query reports the cumulative counts of the number of times each identifier has occurred. In these workloads, the state is the per-identifier count, intentionally small and simple so that we can see the effect of migration rather than associated computation. We consider two variants, an implementation that uses hashmaps for bins ("hash count"), and an optimized implementation that uses dense arrays to remove hashmap computation ("key count").

Each experiment is parameterized by a domain size (the number of distinct keys) and an input rate (in records per second), for which we then vary the number of bins used by Megaphone. We pre-load one instance of each key to avoid state re-allocation at runtime.

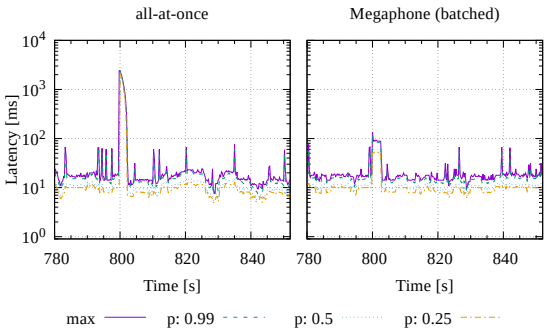


Figure 8: NEXMark query latency for Q4, 4×10^6 requests per second, reconfiguration at 800 s.

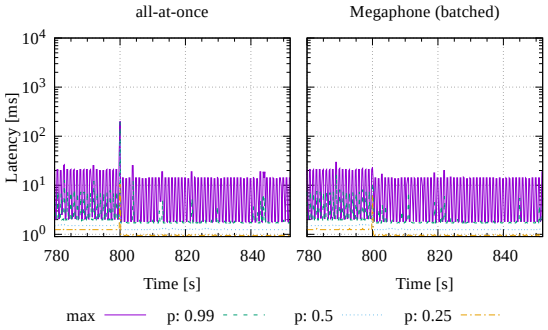


Figure 9: NEXMark query latency for Q5, 4×10^6 requests per second, reconfiguration at 800 s with time dilation.

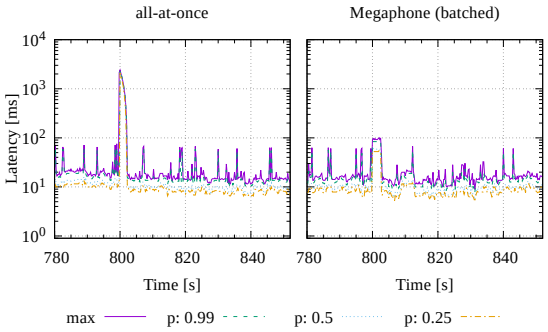


Figure 10: NEXMark query latency for Q6, 4×10^6 requests per second, reconfiguration at 800 s.

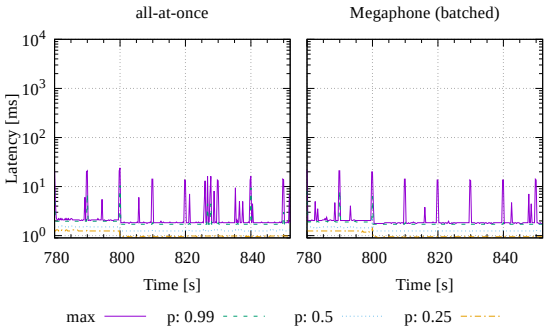


Figure 11: NEXMark query latency for Q7, 4×10^6 requests per second, reconfiguration at 800 s.

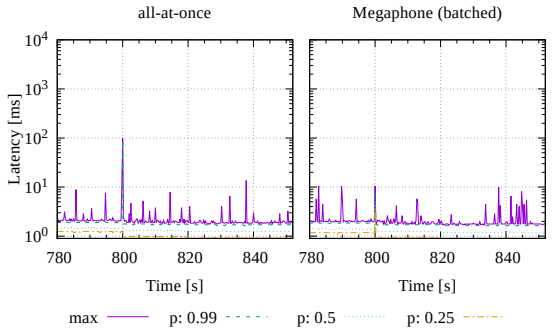
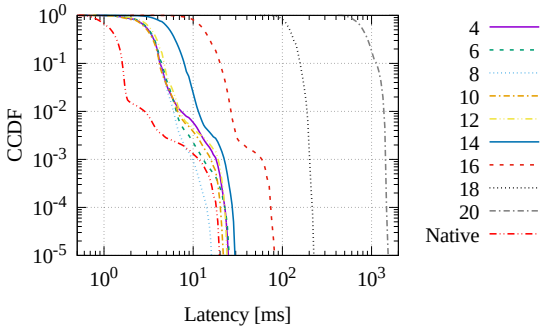


Figure 12: NEXMark query latency for Q8, 4×10^6 requests per second, reconfiguration at 800 s with time dilation.



(a) CCDF of per-record latencies

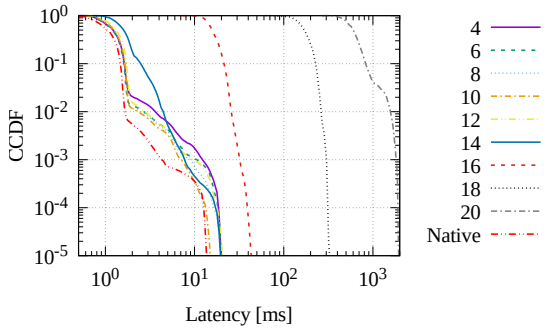
Experiment	90%	99%	99.99%	max
4	4.46	7.60	18.87	25.17
6	4.46	6.55	13.11	26.21
8	4.46	6.03	9.96	16.78
10	4.19	6.82	16.25	23.07
12	4.98	7.08	19.92	24.12
14	8.13	11.53	23.07	30.41
16	20.97	27.26	60.82	83.89
18	159.38	192.94	209.72	226.49
20	1140.85	1409.29	1476.40	1543.50
Native	1.64	2.88	12.06	19.92

(b) Selected percentiles and their latency in ms

Figure 13: Hash-count overhead experiment with 256×10^6 unique keys and an update rate of 4×10^6 per second. Experiment numbers in (a) and (b) indicate log bin count.

Figure 13 shows the complementary cumulative distribution function (CCDF) of per-record latency for the hash-count experiment with 256×10^6 distinct keys and a rate of 4×10^6 updates per second. Figure 14 shows the CCDF of per-record latency for the key-count experiment with 256×10^6 distinct keys and a rate of 4×10^6 updates per second. Figure 15 shows the CCDF of per-record latency for the key-count experiment with 8192×10^6 distinct keys and a rate of 4×10^6 updates per second. Each figure reports measurements for a native timely dataflow implementation, and for Megaphone with geometrically increasing numbers of bins.

For small bin counts, the latencies remain a small constant factor larger than the native implementation, but this increases noticeably once we reach 2^{16} bins. We conclude that while a performance penalty exists, it can be an acceptable trade-off for stateful dataflow reconfiguration. A bin-count parameter of up to 2^{12} leads to largely indistinguishable results, and we will use this number when we need to hold the bin count constant in the rest of the evaluation.



(a) CCDF of per-record latencies

Experiment	90%	99%	99.99%	max
4	1.64	3.67	12.58	19.92
6	1.64	2.75	11.01	20.97
8	1.70	2.49	9.44	19.92
10	1.70	2.36	7.08	15.20
12	1.77	2.88	9.96	20.97
14	2.49	4.46	7.86	19.92
16	22.02	26.21	32.51	46.14
18	234.88	268.44	301.99	335.54
20	838.86	1610.61	1879.05	1946.16
Native	1.51	1.70	4.46	14.16

(b) Selected percentiles and their latency in ms

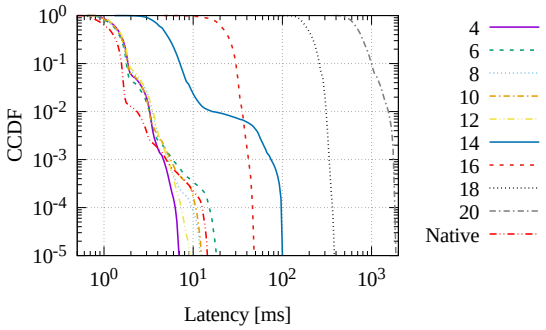
Figure 14: Key-count overhead experiment with 256×10^6 unique keys and an update rate of 4×10^6 per second. Experiment numbers in (a) and (b) indicate log bin count.

5.3 Migration micro-benchmarks

We now use the counting benchmark from the previous section to analyse how various parameters influence the maximum latency and duration of Megaphone during a migration. Specifically,

1. In Section 5.3.1 we evaluate the maximum latency and duration of migration strategies **as the number of bins increases**. We expect Megaphone's maximum latencies to decrease with more bins, without affecting duration.
2. In Section 5.3.2 we evaluate the maximum latency and duration of migration strategies **as the number of distinct keys increases**. We expect all maximum latencies and durations to increase linearly with the amount of maintained state.
3. In Section 5.3.3 we evaluate the maximum latency and duration of migration strategies **as the number of distinct keys and bins increase proportionally**. We expect that with a constant per-bin state size Megaphone will maintain a fixed maximum latency while the duration increases.
4. In Section 5.3.4 we evaluate **the latency under load** during migration and steady-state. We expect a smaller maximum latency for Megaphone migrations.
5. In Section 5.3.5 we evaluate **the memory consumption** during migration. We expect a smaller memory footprint for Megaphone migrations.

Each of our migration experiments largely resembles the shapes seen in Figure 1, where each migration strategy has a well defined *duration* and *maximum latency*. For example, the all-at-once migration strategy has a relatively short duration with a large maximum latency, whereas the bin-at-a-time (*fluid*) migration strategy has a longer duration and lower maximum latency, and the batched migration strategy lies between the two. In these experiments we summarize each migration by the duration of the migration, and the maximum latency observed during the migration.



(a) CCDF of per-record latencies

Experiment	90%	99%	99.99%	max
4	1.90	3.28	4.72	7.34
6	1.84	3.28	6.03	18.87
8	1.84	3.54	5.24	12.58
10	1.84	3.28	5.77	13.11
12	1.90	3.67	5.24	9.44
14	7.34	16.78	75.50	100.66
16	30.41	35.65	41.94	50.33
18	268.44	318.77	335.54	385.88
20	1006.63	1610.61	1811.94	1879.05
Native	1.57	2.36	4.98	14.68

(b) Selected percentiles and their latency in ms

Figure 15: Key-count overhead experiment with 8192×10^6 unique keys and an update rate of 4×10^6 per second. Experiment numbers in (a) and (b) indicate log bin count.

5.3.1 Number of bins vary. We now evaluate the behavior of different migration strategies for varying numbers of bins. As we increase the number of bins we expect to see fluid and batched migration achieve lower maximum latencies, though ideally with relatively unchanged durations. We do not expect to see all-at-once migration behave differently as a function of the number of bins, as it conducts all of its migrations simultaneously.

Holding the rates and bin counts fixed, we will vary the number of bins from 2^4 up to 2^{14} by factors of four. For each configuration, we run for one minute to establish a steady state, and then initiate a migration and continue for one another minute. During this whole time the rate of input records continues uninterrupted.

Figure 16 reports the latency-vs-duration trade-off of the three migration strategies as we vary the number of bins. The connected lines each describe one strategy, and the common shapes describe a common number of bins. We see that all all-at-once migration experiments are in a low duration high latency cluster. Both fluid and batched migration achieve lower maximum latency as we increase the number of bins, without negatively impacting the duration.

5.3.2 Number of keys vary. We now evaluate the behavior of different migration strategies for varying domain sizes. Holding the rates and bin counts fixed, we will vary the number of keys from 256×10^6 up to 8192×10^6 by factors of two. For each configuration, we run for one minute to establish a steady state, and then initiate a migration and continue for one another minute. During this whole time the rate of input records continues uninterrupted.

Figure 17 reports the latency-vs-duration trade-off of the three migration strategies as we vary the number of distinct keys. The connected lines each describe one strategy, and the common shapes describe a common number of distinct keys. We see that for any experiment, all-at-once migration has the highest latency and lowest duration, fluid migration has a lower latency and higher duration, and batched migration often has the best qualities of both.

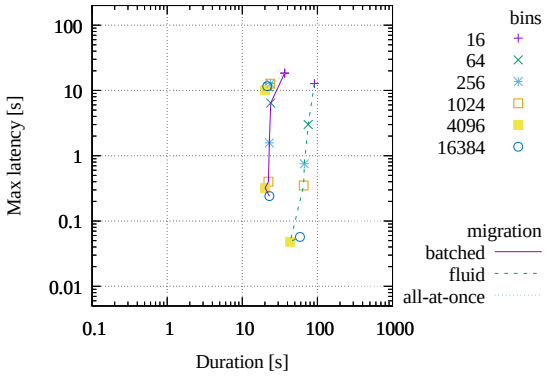


Figure 16: Key-count migration latency vs. duration, varying bin count for a fixed domain of 4096×10^6 keys. The vertical lines indicate that increasing the granularity of migration can reduce maximum latency for fluid and batched migrations without increasing the duration. The all-at-once migration datapoints remain in a cluster independent of the migration granularity.

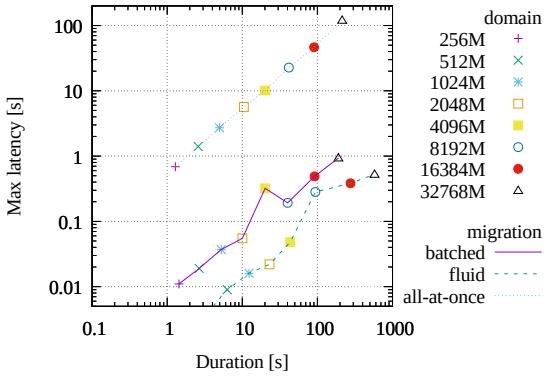


Figure 17: Key-count migration latency vs. duration, varying domain for a fixed rate of 4×10^6. As the domain size increases the migration granularity increases, and the duration and maximum latencies increase proportionally.

5.3.3 Number of keys and bins vary proportionally. In the previous experiments, we either fixed the number of bins or the number of keys while varying the other parameter. In this experiment, we vary both bins and keys together such that the total amount of data per bin stays constant. This maintains a fixed migration granularity, which should have a fixed maximum latency even as the number of keys (and total state) increases. We run the key count experiment and fix the number of keys per bin to 4×10^6. We then increase the domain in steps of powers of two starting at 256×10^6 and increase the number of bins such that the keys per bin stays constant. The maximum domain is 32×10^9 keys.

Figure 18 reports the latency-versus-duration trade-off for the three migration strategies as we increase domain and number of bins while keeping the state per bin constant. The lines describe one migration strategy and the points describe a different configuration. We can observe that for fluid and batched migration the latency stays constant while only the duration increases as we increase the domain. For all-at-once migration, both latency and duration increase.

We conclude that fluid and batched migration bound the latency impact on a computation during a migration while increasing the migration duration, whereas all-at-once migration does not.

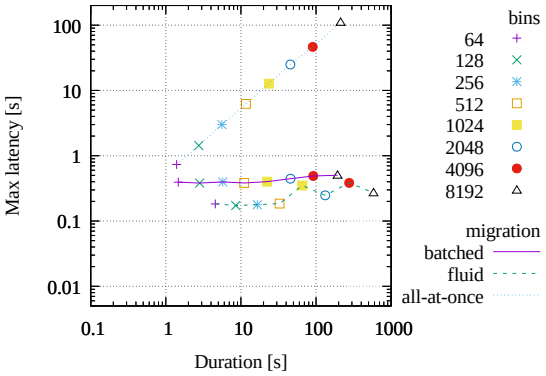


Figure 18: Latency and duration of key-count migrations for fixed state per bin. By holding the granularity of migration fixed, the maximum latencies of fluid and batched migration remain fixed even as the durations of all strategies increase.

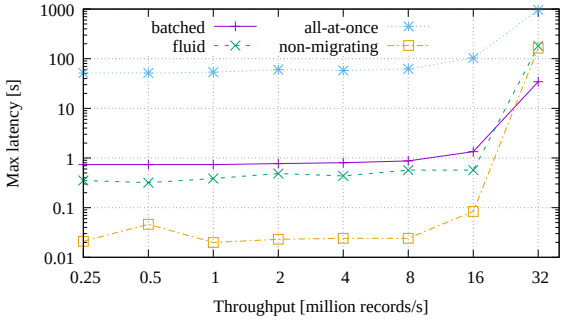


Figure 19: Offered load versus max latency for different migration strategies for key-count. The migration is invariant of the rate up to 16 million records per second.

5.3.4 Throughput versus processing latency. In this experiment, we evaluate what throughput Megaphone can sustain for specific latency targets. As we increase the offered load, we expect the steady-state and migration latency to increase. For a spcific throughput target, we expect the all-at-once migration strategy to show a higher latency than batched, which itself is expected to be higher than fluid.

To analyze the latency, we keep the number of keys and bins constant, at $16\,384 \times 10^6$ and 4096, and vary the offered load from 250×10^3 to 32×10^6 in powers of two. We measure the maximum latency observed during both steady-state and migration for each of the three migration strategies described earlier.

Figure 19 shows maximum latency observed when the system is sustaining a certain throughput. All three migration strategies and non-migrating show a similar pattern: Up to 16×10^6 records per second they do not show a significant increase in latency. At 32×10^6, the latency increases significantly, indicating that the system is now overloaded.

We conclude that the system's latency is mostly throughput-invariant until the system saturates and eventually fails to keep up with its input. Both fluid and batched migration sustain a throughput of up to 4×10^6 per second for a latency target of 1 s: Megaphone's migration strategies can satisfy latency targets 10-100x lower than all-at-once migration with similar throughput.

5.3.5 Memory consumption during migration. In Section 5.3.3 we analyzed the behavior of different migration strategies when increasing the total amount of state in the system while leaving the state per bin constant. Our expectation was that the all-at-once migration strategy would always offer the lowest duration when compared to batched and fluid migrations. Nevertheless, we observe for large amounts of data being migrated the duration for a all-at-once migration is longer than for batched migration.

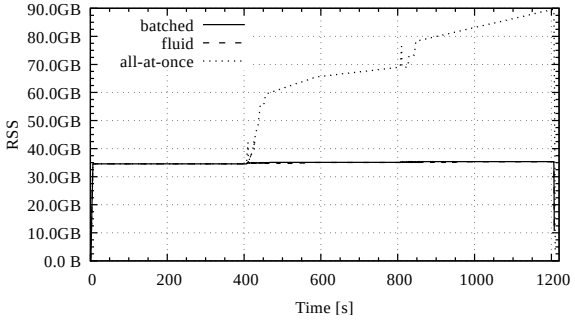


Figure 20: Memory consumption of key-count per process over time for different migration strategies. The fluid and batched strategies require less additional memory in each migration step than the all-at-once migration, which migrates all state at once.

To analyze the cause for this behavior we compared the memory consumption for the three migration strategies over time. We run the key count dataflow with 16×10^9 keys and 4096 bins. We record the resident set size (RSS) as reported by Linux over time per process.

Figure 20 shows the RSS reported by the first timely process for each migration strategy. Batched and fluid migration show a similar memory consumption of 35 GiB in steady state and do not expose a large variance during migration at times 400 s and 800 s. In contrast to that, all-at-once migration shows significant allocations of approximately additional 30 GiB during the migrations.

The experiment gives us evidence that a all-at-once migration causes significant memory spikes in addition to latency spikes. The reason for this is that during a all-at-once migration, each worker extracts and serializes the data to be migrated and enqueues it for the network threads to send. The network thread's send capacity is limited by the network throughput, limiting the throughput at which data can be transferred to the remote host. Batched and fluid migration patterns only perform another migration once the previous is complete and thus provide a simple form of flow-control effectively limiting the amount of temporary state.

6 Conclusion

We presented the design and implementation of Megaphone, which provides efficient, minimally disruptive migration for stream processing systems. Megaphone plans fine-grained migrations using the logical timestamps of the stream processor, and interleaves the migrations with regular streaming dataflow processing. Our evaluation on realistic workloads shows that migration disruption was significantly lower than with prior all-at-once migration strategies.

We implemented Megaphone in timely dataflow, without any changes to the host dataflow system. Megaphone demonstrates that dataflow coordination mechanisms (timestamp frontiers) and dataflow channels themselves are sufficient to implement minimally disruptive migration. Megaphone's source code is available on `https://github.com/strymon-system/megaphone`.

Acknowledgments

We thank Nicolas Hafner for an initial implementation of the NEXMark queries and the anonymous VLDB reviewers for their comments. This work was partly supported by Google, VMware, and the Swiss National Science Foundation. Andrea Lattuada is supported by a Google PhD fellowship and Vasiliki Kalavri by an ETH postdoctoral fellowship.

7 References

[1] Bringing Pokemon GO to life on Google Cloud. https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html.

[2] IBM Streams (accessed: November 2017). https://www.ibm.com/ch-en/marketplace/stream-computing.

[3] New Tweets per second record, and how! https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html.

[4] M. Abadi, F. McSherry, D. G. Murray, and T. L. Rodeheffer. Formal analysis of a distributed algorithm for tracking progress. In D. Beyer and M. Boreale, editors, *Formal Techniques for Distributed Systems - Joint IFIP WG 6.1 International Conference, FMOODS/FORTE 2013, Held as Part of the 8th International Federated Conference on Distributed Computing Techniques, DisCoTec 2013, Florence, Italy, June 3-5, 2013. Proceedings*, volume 7892 of *Lecture Notes in Computer Science*, pages 5–19. Springer, 2013.

[5] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[6] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[7] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 601–613. ACM, 2018.

[8] S. K. Barker, Y. Chi, H. J. Moon, H. Hacigümüs, and P. J. Shenoy. "Cut me some slack": latency-aware live migration for databases. In E. A. Rundensteiner, V. Markl, I. Manolescu, S. Amer-Yahia, F. Naumann, and I. Ari, editors, *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 432–443. ACM, 2012.

[9] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *PVLDB*, 10(12):1718–1729, 2017.

[10] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Data Engineering*, 38(4), 2015.

[11] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *PVLDB*, 4(8):494–505, 2011.

[12] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, New York, NY, USA, 2015. ACM.

[13] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 301–312, New York, NY, USA, 2011. ACM.

[14] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu. Parallel stream processing against workload skewness and variance. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '17, pages 15–26, 2017.

[15] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 725–736, 2013.

[16] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 10(12):1825–1836, 2017.

[17] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 2012.

[18] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[19] M. Hoffmann, F. McSherry, and A. Lattuada. Latency-conscious dataflow reconfiguration. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, page 1. ACM, 2018.

[20] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[21] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao. Chi: A scalable and programmable control plane for distributed stream processing systems. *PVLDB*, 11(10):1303–1316, 2018.

[22] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, Nov 2013.

[23] NEXMark benchmark. http://datalab.cs.pdx.edu/niagaraST/NEXMark.

[24] S. Rajadurai, J. Bosboom, W.-F. Wong, and S. Amarasinghe. Gloss: Seamless live reconfiguration and reoptimization of stream programs. In *ASPLOS*, pages 98–112, 2018.

[25] O. Schiller, N. Cipriani, and B. Mitschang. Prorea: Live database migration for multi-tenant rdbms with snapshot isolation. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 53–64, New York, NY, USA, 2013. ACM.

[26] M. A. Shah, M. A. Shah, S. Chandrasekaran, J. M. Hellerstein, J. M. Hellerstein, S. Ch, S. Ch, M. J. Franklin, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2002.

[27] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark—A Benchmark for Queries over Data Streams DRAFT. Technical report, OGI School of Science & Engineering at OHSU, 2002.

[28] Y. Wu and K. Tan. Chronostream: Elastic stateful stream computation in the cloud. In J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 723–734. IEEE Computer Society, 2015.

[29] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[30] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 431–442, 2004.