Real-time Monitoring and Analysis of Data Streams with Confluo

Anurag Khandelwal
UC Berkeley

Rachit Agarwal
Cornell University

Ion Stoica
UC Berkeley

Abstract

We present Confluo, a system for real-time monitoring and analysis of data streams. Confluo achieves three desirable properties for such systems: (1) high-throughput concurrent writes of millions of data points from multiple data streams; (2) online queries at millisecond timescale; and (3) ad-hoc queries using minimal CPU resources. The key technical contribution in Confluo is a new data structure — Atomic MultiLog — that supports efficiently updating a collection of lock-free concurrent logs as a single atomic operation. Confluo supports the above three properties using Atomic MultiLogs to store data, aggregate statistics and materialized views, along with hardware primitives for efficient atomic updates on Atomic MultiLogs. We use Confluo to build two applications — a network monitoring and diagnosis tool and a time-series database — and demonstrate benefits against state-of-the-art systems for respective applications.

1 Introduction

Many contemporary applications (*e.g.*, end-host based network monitoring [1, 2, 3, 4], IoT and connected homes [5, 6, 7, 8] and datacenter operational services [9, 10, 11, 12]) capture tens of millions of data points per second per backing server. This data is used in online queries for visualization and monitoring purposes, and in offline queries for root-cause analysis and system optimizations. Enabling these applications require real-time monitoring and analysis tools that support:

- **High-throughput concurrent writes** of millions of data points potentially from multiple data streams;
- **Online queries** over raw data, aggregate statistics and materialized views, to support visualization and monitoring at millisecond-level time granularity; and,
- **Ad-hoc queries** by slicing-and-dicing the raw data and materialized views (potentially from multiple streams distributed across multiple backing servers) using minimal CPU resources.

Building systems to *simultaneously* support these functionalities turns out to be a hard problem. The main challenge is that supporting the aforementioned online and offline queries requires updating several data structures — for storing raw data, aggregate statistics and materialized views across multiple attributes [1, 2, 5, 6, 9] — while ingesting data from multiple data streams. Existing state-of-the-art systems can support high-throughput concurrent writes on a single data structure (using highly optimized *log*-based data structures[1]); however, evaluation of three state-of-the-art systems in §6 shows that these systems fail to achieve high concurrency when multiple data structures need to be updated. Thus, these systems can support either high concurrency [1, 2, 5, 8] or expressive queries [6, 9, 13, 14, 15], but not both.

We present Confluo, a system that simultaneously supports high-throughput concurrent writes from multiple data streams, while maintaining updated aggregate statistics and materialized views required for efficient online and offline queries. The key technical contribution of Confluo is a new data structure — Atomic MultiLog. The distinguishing feature of an Atomic MultiLog is that it uses a collection of lock-free concurrent logs (to store raw data, aggregate statistics and materialized views), and uses new techniques to efficiently update this entire collection as a single atomic operation.

Atomic MultiLogs use two core ideas. First, Atomic MultiLogs relax the atomicity guarantees for its individual logs while guaranteeing atomicity only for the entire collection. This allows Atomic MultiLogs to alleviate synchronization overheads necessary to guarantee atomicity for each individual log (Figure 3, §3.3). Using atomic hardware primitives readily available in commodity servers to execute the atomic operation, Atomic MultiLogs are thus able to support millions of reads and writes per second from multiple data streams, while

[1] Design of such log-based data structures requires the assumption on data streams having *write-once* semantics, or data being *append-only*, which is indeed the case for our target applications.

Table 1: Confluo's API. All supported operations are guaranteed to be atomic. See §2 for definitions and detailed discussion.

	API	Description
Writes	`AtomicMultiLog m = create_atomic_multilog(schema)`	Create an Atomic MultiLog with specified `schema`.
	`m.add(stream)`	Add a data `stream` to `m`; `stream` must follow `schema`.
	`m.add_index(attribute)`	Add an index on a fixed-length `attribute`.
	`filterId = m.add_filter(fExpression)`	For all streams in `m`, add filter `fExpression`.
	`aggId = m.add_aggregate(filterId, aFunction)`	Add aggregate `aFunction` on records filtered by `filterId`.
Queries	`m.get_aggregate(aggId, tLo, tHi)`	Get the aggregate `aggId` from time range `(tLo, tHi)`.
	`triggerId = m.install_trigger(aggId,conditional,period)`	Install a trigger over aggregate `aggId` that evaluates `conditional` every `period`, and listen to trigger alert.
	`Iterator<Record> it = m.query(fExpression, tLo, tHi)`	Filter records matching `fExpression` during time `(tLo,tHi)` using pre-existing filter or attribute indexes.
	`m.remove_index(attribute)`, `m.remove_filter(filterId)`, `m.remove_aggregate(aggId)`, `m.uninstall_trigger(triggerId)`	Remove or uninstall specified index, filter, aggregate or trigger.

maintaining updated aggregated statistics and materialized views for efficiently supporting online and ad-hoc queries. To achieve the above, similar to log-based data structures, Atomic MultiLogs exploit the append-only nature of data in our target applications.

The second idea in Atomic MultiLog design uses the observation that data in our target applications is not only append-only, but also has an additional structure — each attribute has an upper limit on its width (number of bits used to represent the attribute). For instance, networks typically use fixed-width packet header fields; and, data generated in IoT sensors and datacenter operational metrics uses a maximum number of bits for each attribute (32-bit sensor and temperature readings, 64-bit timestamps, CPU and memory statistics, etc.). Upper bound on attribute width allows Atomic MultiLogs to use specialized mechanisms to further improve the throughput for ingesting data and for online queries (§3).

Confluo incorporates Atomic MultiLogs into an end-to-end system that supports distributed snapshot (useful for offline queries), fault tolerance, data durability and data archival (§4). We implement two applications on top of Confluo to demonstrate its generality — an end-host based network monitoring and diagnosis framework and a time-series database. Our evaluation in §5 suggests that Confluo outperforms the state-of-the-art systems for respective applications either in terms of supported functionality or in terms of performance (sometimes by as much as an order of magnitude).

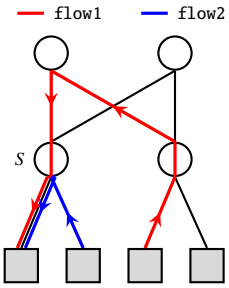
The current implementation of Confluo has two limitations. First, it does not provide transactional guarantees. This allows Confluo to achieve significantly higher throughput than systems that provide transactional guarantees (§5), a tradeoff that may be favorable for several applications [1, 2, 9, 5]. Second, Confluo currently supports online and offline queries only on attributes whose width is upper bounded. As discussed above, data attributes in many applications do have this property; however, Confluo can still be used for applications that have arbitrary sized attributes. For instance, our implementation of a distributed messaging system (similar to Apache Kafka [16] and Amazon Kinesis [17], Appendix A.3) achieves 4-5× higher write throughput and 5-10× higher read throughput than Apache Kafka [16].

In summary, this paper makes three contributions:

- Design and implementation of Confluo, a system that simultaneously supports the three desirable properties for real-time monitoring and analysis of data streams: (1) high-throughput concurrent writes of millions of data points from multiple streams; (2) millisecond-scale online queries; and, (3) ad-hoc queries using minimal CPU resources.
- Design of Atomic MultiLog, a data structure that supports efficiently updating a collection of lock-free concurrent logs as a single atomic operation.
- Implementation and evaluation of two applications on top of Confluo: a network monitoring and diagnosis framework and a time-series database.

2 Confluo Data Model and API

Confluo operates on data streams. Each stream comprises of records, each of which follows a pre-defined *schema* over a collection of typed attributes. Attributes could be *bounded-width* or *variable-width*. An attribute is said to have bounded-width if all records in a single stream use some maximum number of bits to represent that attribute; this includes primitive data types such as binary, integral or floating-point values, or domain-specific types such as IP addresses, ports, sensor readings, etc. We call an attribute variable-width otherwise. Confluo also requires each record in the stream to have a timestamp attribute; if the application does not assign timestamps, Confluo internally assigns one during the write operation.



Scenario	Monitoring	Diagnosis
rate(`flow1`) + rate(`flow2`) > bandwidth, Priority of packets equal in `flow1`, `flow2` Packet drops for both `flow1` and `flow2`.	**Filter** all retransmitted TCP pkts, **Trigger** alert if `count>threshold` in last `timeInterval`	$c_1 \leftarrow$ `count(flow1` retransmitted pkts), $c_2 \leftarrow$ `count(flow2` retransmitted pkts), $p_1 \leftarrow$ `flow1` priority, $p_2 \leftarrow$ `flow2` priority in last 1s, check if $c_1 \approx c_2 > 0$ && $p_1 = p_2$
rate(`flow1`) + rate(`flow2`) > bandwidth, Priority of packets in `flow1` < `flow2`, Packet drops for `flow1` only.	**Filter** all retransmitted TCP pkts, **Trigger** alert if `count>threshold` in last `timeInterval`	c_1, c_2, p_1, $p_2 \rightarrow$ Same as above in last 1s, check if: $c_1 \approx 0$ && $c_2 > 0$ && $p_1 < p_2$ or, $c_2 \approx 0$ && $c_1 > 0$ && $p_2 < p_1$
rate(`flow1`) + rate(`flow2`) < bandwidth, Bug at S drops based on packet timing, Packet drops for both `flow1` and `flow2`.	**Filter** all TCP retransmission pkts, **Trigger** alert if `count>threshold` in last `timeInterval`	$t_i \leftarrow$ Retransmitted pkt timestamps, $\delta_i \leftarrow t_i - t_{i-1}$, check if standard dev. $\sigma(\delta_i) <$ threshold

Figure 1: Network Monitoring & Diagnosis Example. (left) A switch S with two incident flows; S may drop packets for one or both flows due to various scenarios. (right) Confluo queries to monitor packet drops and diagnose causes for packet drops at S.

Table 2: Elements of Confluo filters, aggregates and triggers.

	Operator	Examples
Relational	**Equality**	`dstPort=80`
	Range	`cpuUtil>80%`, `srcIP in 10.1.3.0/24`
	Wildcard	`dstIP like 192.*.*.1`
Boolean	**Conjunction**	`volt>200V && temp>100F`
	Disjunction	`cpuUtil>80% || memAvail<10%`
	Negation	`transportProtocol!=TCP`
Aggregate	**AVG**	`AVG(latency)>100ms`
	COUNT, SUM	`COUNT(error)>10`, `SUM(pktSize)<1KB`
	MAX, MIN	`MIN(volt)<100V`, `MAX(temp)>100F`

API. Table 1 outlines Confluo API. Applications can create an Atomic MultiLog with a pre-specified `schema`, and `add` to it data streams that conform to the `schema`.

To support queries, applications can add an `index` for individual attributes in the `schema`. Confluo also employs a match-action language with three main elements: `filter, aggregate` and `trigger`. A Confluo filter is an expression `fExpression` comprising of relational and boolean operators (Table 2) over arbitrary subset of fixed-length attributes, and identifies records that match `fExpression`. A Confluo aggregate evaluates a computable function (Table 2) on an attribute for all records that match a certain filter expression. Finally, a Confluo trigger is a boolean conditional (*e.g.*, `<, >,` =, etc.) evaluated over a Confluo aggregate. Confluo supports indexes, filters, aggregates and triggers only on bounded-width attributes in the `schema`. We describe the design and implementation of Confluo indexes, filters, aggregates and triggers in §3.3 and §3.4, but make a note here that once added, each of these are evaluated and updated upon arrival of each new batch of data records.

2.1 End-to-end Pipeline: An Example

We discuss an example from network monitoring and diagnosis application to further illustrate the Confluo API.

Consider the toy example in Figure 1 where switch S in a data center network drops packets. These drops can happen for multiple reasons, and can lead to different packet loss patterns. If S is congested, then either both flows experience packet drops, or only one of them does (if, for example, the other flow has higher priority). Even in the absence of congestion, S may drop arbitrary subset of packets due to faulty interfaces or software bugs that are induced by packet trajectory, header information or packet timing [18, 2]. Accurate and rapid diagnosis of such network events can enable network operators to take appropriate actions to minimize performance degradation (*e.g.*, in Figure 1, rate-limit or schedule the corresponding flow(s) in case of congestion, re-route packets in case of faulty hardware or software bugs).

Figure 1 (right) outlines how Confluo can detect and diagnose any of the three scenarios described above. Confluo filters, aggregates and triggers allow detecting spurious packet losses efficiently by tracking TCP packets and periodically evaluating a trigger to check if the retransmission count exceeds a threshold (and raise an alarm if it does). For diagnosis, Confluo's `query` operation allows filtering precisely those packets that match the `fExpression` that led to the trigger raising an alarm.

3 Confluo Design

We briefly recall the design of log-based data structures (§3.1) and then give a high-level overview of Confluo design (§3.2). We then provide details on Atomic MultiLogs (§3.3) and Confluo's query framework (§3.4).

3.1 Background

We briefly review two concepts from prior work that will be useful in succinctly describing Confluo design.

Atomic Hardware Primitives. Most modern CPU architectures support a variety of atomic instructions. Confluo will use four such instructions: `AtomicLoad`, `AtomicStore`, `FetchAndAdd` and `CompareAndSwap`. All four instructions operate on 64 bit operands. The first two permit atomically reading from and writing to memory locations. The `FetchAndAdd` instruction atomically obtains the value at a memory location and increments it. Finally, the `CompareAndSwap` atomically compares the

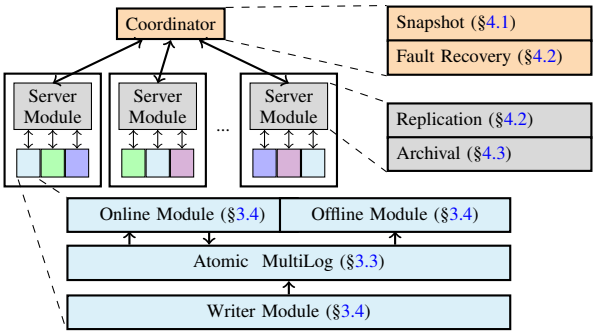


Figure 2: High-level Confluo Architecture (§3.2).

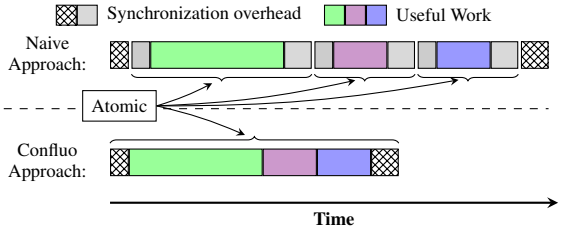


Figure 3: Confluo relaxes the atomicity guarantees of individual logs, while guaranteeing atomicity only for end-to-end Atomic MultiLog operations. Different colors correspond to operations on different logs.

value at a memory location to a given value, and only if they are the same, modifies the value at the memory location to a new specified value.

Logs. There has been a tremendous amount of prior work on design of efficient, lock-free concurrent logs [19, 20, 21, 22] that exploit the append-only nature in many applications to support high-throughput writes. Intuitively, each log maintains a "writeTail" that marks the end of the log. Every new append operation increments the writeTail by the number of bytes to be written, and then writes to the log. Using the above hardware primitives to atomically increment the writeTail, these log based data structure support extremely high write rates.

It is rather trivial to show that by additionally maintaining a "readTail" that marks the end of completed append operations (and thus, always lags behind the writeTail) and by carefully updating the readTail, one can guarantee *atomicity* for concurrent reads and writes on a single log (see Appendix A.1 for a simple, formal proof). Using atomic hardware primitives to update both readTail and writeTail, it is then possible to achieve high throughput for concurrent reads *and* writes for such logs.

3.2 Confluo Design Overview

Confluo architecture comprises a central coordinator and a set of backing servers[2] (see Figure 2). Each backing server may host one or more Atomic MultiLogs, depending on the `schema` of streams backed at that server (recall that all streams added to an Atomic MultiLog must follow a fixed `schema`). Each server runs a server module that handles replication and durability of Atomic MultiLogs hosted at that server. The central coordinator serves two main functions: (1) periodically obtaining snapshots for data distributed across various Atomic MultiLogs; and, (2) detecting and resolving failures across storage servers via chain replication protocol. We describe server module and coordinator functionalities in §4.

Figure 4 shows an example Atomic MultiLog— it uses a number of lock-free concurrent logs to store the raw data in the streams (StreamLog), materialized views on certain attributes (IndexLog), pre-defined filters (FilterLog) and aggregates (AggregateLog). We describe the design of these logs in depth in §3.3.1.

A writer module writes the data stream records to StreamLog; upon each write, all logs are updated as a single atomic operation. However, while naïvely updating each individual lock-free concurrent log guarantees atomicity for individual logs, it requires significant synchronization overheads (Figure 3). Atomic MultiLog avoids this synchronization overhead by relaxing the atomicity guarantees for individual logs while guaranteeing atomicity only for end-to-end Atomic MultiLog operations. To achieve this, Atomic MultiLog exploit the append-only nature of data in our target applications — it generalizes the "readTail" and "writeTail" markers in log-based data structures (§3.1) to "globalReadTail" and "globalWriteTail" for Atomic MultiLog; by carefully updating the global read and write tail markers, Atomic MultiLog is able to guarantee atomicity for end-to-end Atomic MultiLog operations. We describe this in §3.3.2.

The online module supports millisecond-scale online queries on Atomic MultiLogs using FilterLog, AggregateLog and triggers on AggregateLog (defined in §2). Finally, an offline module uses IndexLog and StreamLog to support resource-efficient ad-hoc queries. These modules are described in §3.4.

3.3 Atomic MultiLog

We now describe Atomic MultiLog (§3.3.1), and the techniques that allow an Atomic MultiLog to update the entire collection as a single atomic operation (§3.3.2).

3.3.1 Data Structures

An Atomic MultiLog uses a collection of concurrent lock-free logs to store data streams, indexes, aggregates and filters defined in §2 (see Figure 4).

StreamLog. This log stores the raw data for all streams added to the Atomic MultiLog. Each record in a StreamLog has an offset, which is used as a unique reference

[2]Confluo leaves it up to the upstream application to decide the server that backs each data stream.

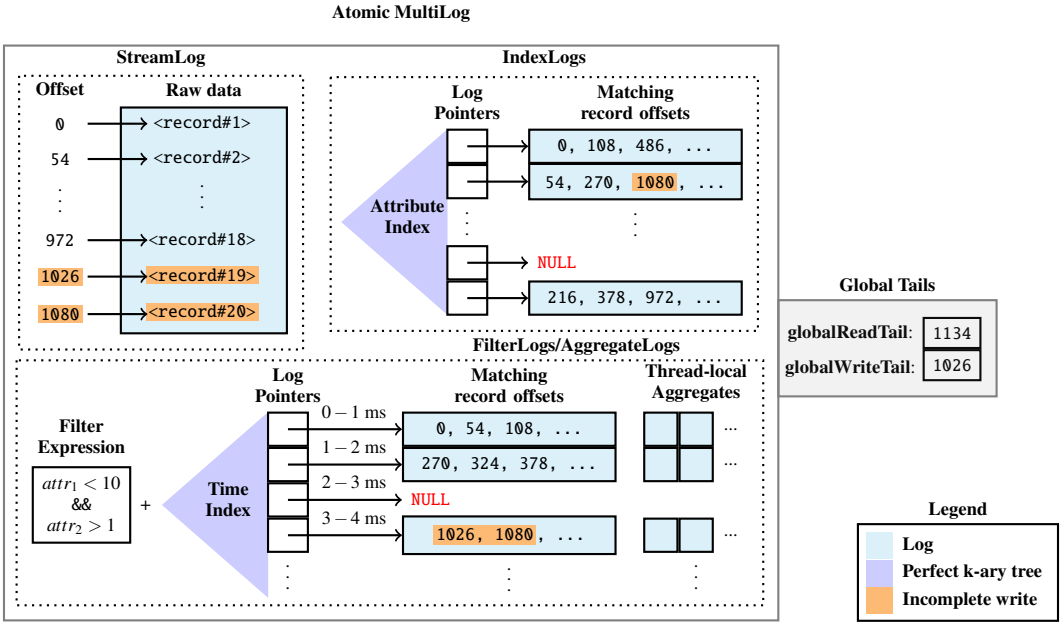


Figure 4: The Atomic MultiLog uses a collection of concurrent lock-free logs to store data streams, indexes, aggregates and filters (as defined in §2) and efficiently updates these data structures as a single atomic operation as new data arrives. See §3.3 for details.

to the record across all data structures within the Atomic MultiLog. We will discuss in §3.4 how this simplifies guaranteeing atomicity for operations that span multiple data structures within the Atomic MultiLog.

IndexLog. An Atomic MultiLog stores an IndexLog for each indexed attribute, that maps each unique attribute value to corresponding StreamLog records. IndexLogs efficiently support concurrent, lock-free insertions and lookups using two main ideas.

First, the upper bound on attribute width enables IndexLogs to use a perfect k-ary tree [23] (referred to as an attribute index in Figure 4) for high-throughput insertions upon new data arrival. Specifically, an attribute with a width upper bound of n-bits is indexed using a k-ary tree with a depth of $\lceil \frac{n}{log_2 k} \rceil$ nodes, where each node indexes $log_2 k$ bits of the attribute (see Figure 5 for an example). The use of a perfect k-ary tree greatly simplifies the write path. All child pointers in a k-ary tree node initially point to `NULL`. When a new attribute value is indexed, all unallocated nodes along the path corresponding to the attribute value are allocated. This is where an IndexLog uses the second idea — since the workload is append-only, StreamLog offsets for attribute value to record mapping are also append-only; thus, traditional lock-free concurrent logs can be used to store this mapping at the leaves of the k-ary tree.

Conflicts among concurrent attribute index nodes and log allocations are resolved using the `CompareAndSwap` instruction, thus alleviating the need for locks. Subsequent records with the same attribute value are indexed

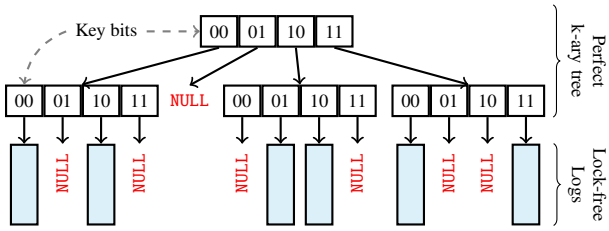


Figure 5: IndexLog with k=4 and n=4. Each node in the tree (depth=2) has k=4 children and indexes 2 bits of the key.

by traversing the tree to the relevant leaf, and appending the record's offset to the log. To evaluate range queries on the index, Confluo identifies the sub-tree corresponding to the attribute range; the final result is then the union of record offsets across logs in the sub-tree leaves.

FilterLog. A FilterLog is simply a filter expression, and a time-indexed collection of logs that store references to records that match the expression (bucketed along 1ms intervals). The logs corresponding to different 1ms intervals are indexed using a perfect k-ary tree, similar to Confluo attribute indexes.

AggregateLog. Similar to FilterLogs, an AggregateLog employs a perfect k-ary tree to index aggregates that match a filter expression across 1ms time buckets. However, consistent updates on aggregate values is slightly more challenging — it requires reading the most recent version, modifying it, and writing it back. Maintaining a single concurrent log for aggregates requires handling complex race conditions to guarantee atomicity. Con-

fluo instead maintains a collection of *thread-local* logs, with each writer thread executing a read-modify-write operation on its own aggregate log independent of other threads. The latest version of an aggregate is obtained by combining the most recent thread-local aggregate values from individual logs.

3.3.2 Atomic Operations on Collection of Logs

End-to-end operations on Atomic MultiLog may require updating multiple logs across StreamLog, IndexLogs and FilterLogs. Even if individual logs can support atomic operations, an end-to-end operation on Atomic MultiLog is not guaranteed to be atomic by default. It is possible to extend log readTail/writeTail mechanism to guarantee atomicity for Atomic MultiLog operations; however, this requires resolving two challenges.

First, in order to guarantee total order for Atomic MultiLog operations, its component logs must agree on an ordering scheme. Confluo uses StreamLog as single source of ground truth, and designates StreamLog readTail and writeTail as globalReadTail and globalWriteTail for the Atomic MultiLog. Write operations begin by atomically incrementing the globalWriteTail using the `FetchAndAdd` instruction. This atomic operation resolves potential write-write conflicts, since it assigns a unique StreamLog offset to each write. The write operation then updates all relevant logs in Atomic MultiLog, and end by atomically updating globalReadTail, making the data available to subsequent operations.

The globalReadTail imposes a *total order* among write operations on the Atomic MultiLog based on StreamLog offsets: Confluo only permits a write operation to update the globalReadTail value after all write operations writing at smaller StreamLog offsets have updated the globalReadTail, via repeated `CompareAndSwap` instruction attempts. This ensures that there are no "holes" in the StreamLog, and allows Confluo to ensure correctness for queries via a simple globalReadTail check. In particular, queries first atomically obtain the globalReadTail value using the `AtomicLoad` instruction, and only access data and references for records that lie within the globalReadTail in the StreamLog. Note that since queries do not modify the globalReadTail, they cannot conflict with other queries or write operations in Confluo.

The second challenge lies in preserving atomicity for operations on Confluo aggregates, since they are not associated with any single record that lies within or outside the globalReadTail. To this end, aggregate values in the AggregateLogs are versioned with the StreamLog offset of the write operation that updates it. To get the final aggregate value, Confluo obtains the aggregate with the largest version smaller than the current globalReadTail value for each of the thread-local aggregates. Since each Confluo writer thread modifies its own local aggregate, and queries on aggregates only access versions that are smaller than the globalReadTail, operations on predefined aggregates are rendered atomic.

While the operations above enable end-to-end atomicity for Atomic MultiLog operations, we note that readTail updates for each individual log in the Atomic MultiLog may add up to a non-trivial amount of overhead (Figure 3). Confluo alleviates this overhead by observing that in any Atomic MultiLog operation, the globalReadTail is only updated after each of the individual log readTails are updated. Therefore, any query that passes the globalReadTail check trivially passes the individual readTail checks, obviating the need for maintaining individual readTails. Removing individual log readTails relaxes ordering guarantees for writes on them, while enforcing it only for an end-to-end operation. This significantly reduces contention among concurrent operations.

3.4 Confluo Query Framework

We now describe Confluo's query framework that comprises of a writer, an online and an offline module.

Writer Module. Data streams added to an Atomic MultiLog are ingested by Confluo's writer module, that comprises a pool of writer threads. Each thread writes batches of records from data streams to the MultiLog, updating all relevant logs atomically as outlined in §3.3.2.

Online Module. The online module in Confluo is responsible for providing monitoring functionality, which primarily involves the trigger evaluation framework. Confluo maintains a dedicated monitor thread for each Atomic MultiLog for trigger evaluation. When a new trigger is installed (§2), Confluo transparently adds relevant aggregates to the Atomic MultiLog. Since these aggregates are updated for every record, trigger evaluation itself involves very little work. The monitor thread wakes up at periodic intervals, and first obtains relevant aggregates for time-intervals since the trigger was last evaluated. It then checks if the trigger predicate (*e.g.*, $\texttt{sum(attr}_1\texttt{)>1000}$) is satisfied, and if so, relays an alert to the upstream application.

Offline Module. Confluo's offline module serves ad-hoc queries on records in the Atomic MultiLog. Recall from Table 1 that Confluo allows a query to provide a filter expression `fExpression` as well as a time range. If there already exists a filter `fExpression`, the query execution is fairly straightforward — since FilterLogs are time-indexed (Figure 4), Confluo simply looks up the FilterLog(s) to extract record offsets corresponding to the specified time interval, drops the offsets that are greater than the globalReadTail value, and returns records corresponding to the remaining offsets. Confluo allows nested queries; an application can apply additional filters on these records or obtain attribute aggregates for them.

If a filter for `fExpression` specified in the query does not already exist, Confluo first performs IndexLog lookups for individual attributes in the filter expression (§3.3), and then combine their results based on the boolean operators in the expression (Table 2). This can be an expensive operation; to that end, Confluo uses several optimizations. For instance, Confluo first converts the filter expression to its canonical disjunctive normal form (DNF) [24], where the resulting filter expression is a disjunction (`OR`) of conjunction (`AND`) clauses. The DNF form yields the most selective filter sub-expressions in its conjunction clauses. In order to minimize the number of record references scanned for a specific conjunction clause, Confluo uses the tail value for individual attributes IndexLog as an estimate of their selectivity; Confluo then evaluates the conjunction clause by scanning through IndexLog for the most selective attribute, dropping all records that are larger than the globalReadTail, or do not satisfy the remaining predicates in the clause. The results for individual conjunction clauses are combined using a simple set union for the disjunction operator.

4 Confluo System Properties

We now describe Confluo's distributed atomic snapshot, replication, data durability and archival mechanisms.

4.1 Distributed Atomic Snapshots

If all Confluo operations are confined to a single Atomic MultiLog, the composability property of atomicity [20] guarantees that Confluo as an end-to-end system guarantees atomicity. However, some applications may require a consistent *snapshot* for data streams across Atomic MultiLogs (at or across backing servers).

Existing systems that support consistent distributed snapshots either employ a centralized sequencer to order all writes to the system (*e.g.*, transaction managers in databases [25, 26, 27], log sequencers [13, 14, 15]) making it easy to obtain global snapshots, or employ algorithms that provide weak consistency guarantees (*e.g.*, causal consistency [28]). However, neither is acceptable for Confluo— the former limits write throughput considerably (see §6.2) while the latter provides weaker semantics than Confluo's single-server operations.

To efficiently support *atomic* distributed snapshots, we note that (1) totally ordering writes across servers, while necessary for supporting arbitrary distributed transactions, is an overkill for just supporting atomic snapshots, and (2) write-once semantics in Confluo can greatly simplify snapshot for individual Atomic MultiLogs[3]. Although naively reading readTails at individual Atomic MultiLogs may not produce an atomic snapshot for a collection of Atomic MultiLogs (see Figure 6 (left)), it does hint towards a possible solution.

In particular, atomic distributed snapshot in Confluo reduces to the widely studied problem of obtaining the snapshot of n atomic registers in shared memory architectures [29, 30, 31]. These approaches, however, rely on multiple iterations of register reads with large theoretical bounds on iteration counts. While acceptable in shared memory architectures, where reads are relatively cheap, they are impractical for distributed settings since reads over the network are expensive.

Confluo atomic distributed snapshot algorithm exploits the observation that any snapshot can be rendered atomic by delaying certain writes that would otherwise break atomicity for the snapshot. For instance, in Figure 6 (left), if we delay the completion of writes W1 and W5 until after the last globalReadTail is read on Atomic MultiLog #4 (indicated by the dashed line), the original snapshot becomes atomic, since W1 and W5 now do take effect after W3 and W7 (Figure 6, right).

Algorithm 1 outlines the steps involved in obtaining an atomic snapshot. The coordinator first sends out `FreezeReadTail` requests to all Atomic MultiLogs in the system in parallel. The Atomic MultiLogs then freeze and return the value of their readTail atomically via a `CompareAndSwap` instruction. This temporarily prevents write operations across all Atomic MultiLogs from completing, since they are unable to update the Atomic MultiLog readTails, but does not affect Confluo queries. Once the coordinator receives all the readTails, it issues `UnfreezeReadTail` requests to all the Atomic MultiLogs, causing them to unfreeze their readTail using another `CompareAndSwap` instruction. They then send an acknowledgement to the coordinator, permitting pending writes to complete at once. Since the first `UnfreezeReadTail` message is only sent out after the last Atomic MultiLog readTail has been read, all writes that would conflict with the snapshot are delayed until after the snapshot has been obtained.

The coordinator periodically executes the snapshot algorithm and generates a snapshot vector comprising Atomic MultiLog readTail values, which is used by outstanding queries that need it. Based on application requirements, queries can either wait for the next execution of the snapshot algorithm to obtain the latest snapshot, or obtain a slightly stale but consistent snapshot immediately. Once the query obtains the snapshot vector, it performs the required analysis across different Atomic MultiLogs, ignoring records that lie beyond the snapshot readTail in their respective Atomic MultiLog.

Note that while the readTails remain frozen, write operations can still update the StreamLog, IndexLogs, FilterLogs and AggregateLogs, but wait for the readTail to

[3] Atomic snapshot of any Atomic MultiLog is trivially obtained by reading its globalReadTail

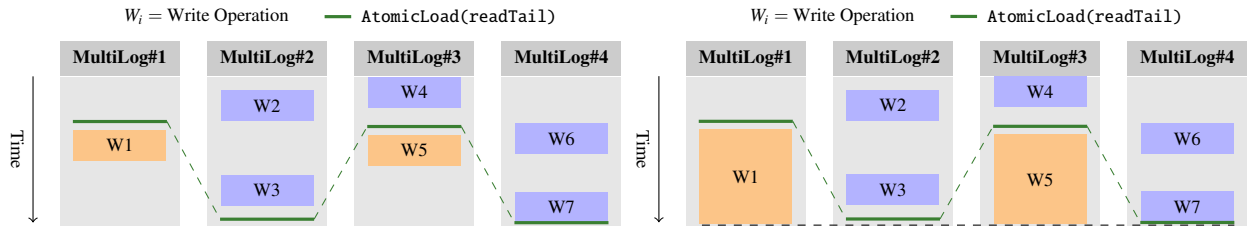


Figure 6: Simply obtaining the (global) readTails for a collection Atomic MultiLogs can yield inconsistent snapshots. This is depicted in (left), where `AtomicLoad` on readTails at different MuliLogs are skewed in time; as a result, W1 and W5 appear to occur after W3 and W7, which is inconsistent. We can render the same snapshot consistent by delaying the completion of W1 and W5 until *after* the last `AtomicLoad` operation (on Atomic MultiLog #4 in this example), as shown in (right).

Algorithm 1 Distributed Atomic Snapshot

Obtains the snapshot vector (Atomic MultiLog readTails).

At Coordinator:

1: `snapshotVector` $\leftarrow \emptyset$
2: Broadcast `FreezeReadTail` requests to all Atomic MultiLogs
3: **for each** `mLog` in `multiLogSet` **do**
4: Receive `readTail` from `mLog` & add to `snapshotVector`
5: **end for**
6: Broadcast `UnfreezeReadTail` requests to all Atomic MultiLogs
7: **for each** Atomic MultiLogmLog **do**
8: Wait for `ACK` from `mLog`
9: **end for**
10: **return** `snapshotVector`

At Each Atomic MultiLog:

On receiving `FreezeReadTail` request

1: Atomically read and freeze `readTail` using `CompareAndSwap`
2: Send `readTail` value to Coordinator

On receiving `UnfreezeReadTail` request

1: Atomically unfreeze `readTail` using `CompareAndSwap`
2: Send `ACK` to Coordinator

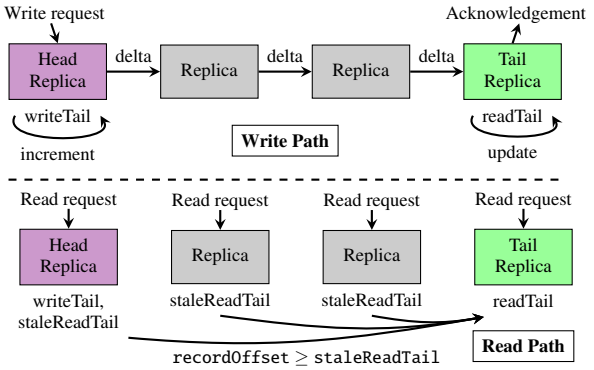


Figure 7: Illustration of Chain replication in Confluo (§4.2).

unfreeze before they finish. This only prevents the effect of these writes from being visible to queries for this duration (i.e., one network round-trip time). As such, write throughput in Confluo is minimally impacted, but write latencies can increase slightly for a short duration. Confluo strikes a balance between the impact on write latencies and the wait-time for queries to obtain a snapshot by tuning the periodicity of snapshots.

4.2 Replication

Confluo coordinator is responsible for detecting failed data sinks (via periodic heartbeat messages from data sinks) and managing replicated Atomic MultiLogs. Confluo makes simple adaptation to read and write paths in the chain replication protocol [32] (shown in Figure 7) by exploiting the append-only nature of Atomic MultiLogs; this allows Confluo to achieve high read and write concurrency while preserving its consistency guarantees. We briefly describe these adaptations below.

The chain replication protocol requires the write operation to be performed only at the head of the replica chain. Confluo observes that once the writeTail at the head has been updated, all replicas along the chain will use the same writeTail for that write. Thus, it suffices to maintain a single writeTail at the head replica; this reduces the write-write contention at other replicas in the chain. Similarly, since a write operation is only considered complete once its effect reaches the tail replica, Confluo maintains a single readTail for the chain at the tail replica. Intermediate replicas in the chain maintain a stale copy of the chain's readTail at each replica, which is periodically updated via multicasts from the tail replica. As a result, intermediate replicas in the chain can serve read requests if the query only accesses records with StreamLog offsets smaller than the stale readTail, but must otherwise forward the requests to the tail replica (similar to CRAQ [33]). This allows Confluo to preserve the atomicity guarantees while (partially) distributing the read load across Atomic MultiLog replicas.

4.3 Data Durability and Archival

In order to provide the illusion of an infinite log, Confluo logs are implemented as a continuously growing collection of large blocks. These blocks are memory mapped for StreamLog, while IndexLog, FilterLog and AggregateLog blocks are drawn from a memory pool.

Durability. Confluo supports both `durable` and `non_durable` write operations on StreamLog; Confluo does not persist IndexLog, FilterLog and AggregateLog, but instead flushes their data to disk during periodic checkpoints. During crash recovery, lost data for all data structures other than StreamLog is reconstructed by replaying the StreamLog beginning from the last checkpoint. For durable writes on StreamLog, memory mapped pages modified by a write are flushed to SSD before the write is considered complete. Confluo also supports writes without `durable` mode, where the StreamLog pages are flushed periodically rather than for every write; as we show in §5, this mode trades off durability guarantees for higher write throughput.

Archival. Storing each of StreamLog, IndexLogs, FilterLogs and AggregateLogs in Confluo over long time periods would lead to tremendous amount of stored data. Several approaches exist for data archival, *e.g.*, periodically summarizing older data with aggregated statistics, log compression [34, 35, 36], compaction [37, 38, 39], or even pushing data to cold storage at periodic intervals. Confluo could use any of these approaches; however, the immutability of data in Confluo (before readTail) allows contention-free archival from SSD to cold storage.

5 Confluo Performance

Confluo prototype is implemented in $\sim$ 20K lines of C++. We now evaluate the performance of Atomic MultiLog (§5.1), and core Confluo system components (§5.2).

5.1 Atomic MultiLog Performance

We start by evaluating the Atomic MultiLog performance using microbenchmarks on a server with 2×12-core 2.30GHz Xeon CPUs and 252GB RAM. The workload uses a stream of 64B records with 20 attributes of widths varying from 1B to 6B. Given the small record size, we found the system overheads to be significant for ingesting each record independently; we thus use a modest batch size of 32 records for all our experiments.

Confluo supports high-throughput concurrent writes. Figure 8(a) shows that, without any filters or indexes, a single Atomic MultiLog ingests as many as $\sim$ 25 million records per second using a single core. The ingestion rate degrades gracefully as more attribute indexes and/or pre-defined filters are added to the Atomic MultiLog; in fact, even with 4 indexes and 64 filters, a single Atomic MultiLog is able to ingest 2.4 million records per second using a single core. As one would expect, the degradation in ingestion rate is nearly linear with the number of indexes (that incur fixed overhead per record) and is sublinear for filters (that incur fixed overheads for records that match the filter, and negligible overhead otherwise). Figure 8(b) and 8(c) show that even with multiple indexes and filters, the ingestion rate scales well with the number of cores, thanks to the lock-free synchronization for concurrent writes in Atomic MultiLog.

It turns out that IndexLog update overheads are higher than any other operation in Confluo; thus, performance trends with indexes are similar to those in Figure 8(c). Hence, unless mentioned otherwise, the rest of the results on write performance use filters and aggregates only.

Confluo evaluates triggers at hardware latency. Recall from §3.3 that Confluo evaluates triggers over predefined aggregates, making trigger evaluation extremely cheap. Figure 8(d) shows that even when Confluo evaluates 1000 triggers at 1ms time intervals, the CPU utilization remains $< 4\%$ of a single core. This is because a single trigger evaluation incurs roughly 70ns latency, with latency increasing to 70μs for 1000 triggers[4].

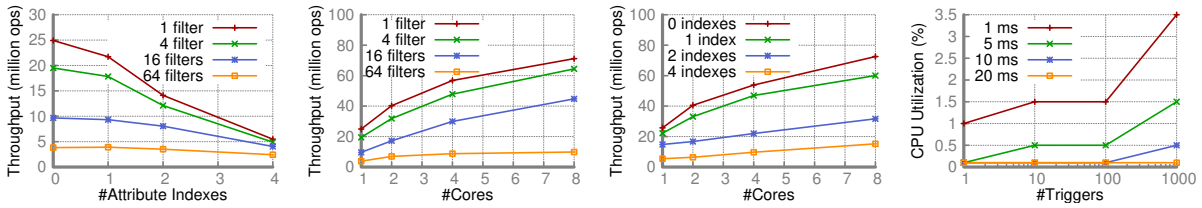
5.2 System Performance

We now evaluate the performance of Confluo over a distributed cluster comprising of Amazon EC2 r3.8xlarge instances with 244GB RAM, 16 CPU cores, and a RAID0 array with 2×300GB SSDs. Unless mentioned otherwise, the workload uses streams of 64B records and each Confluo server uses a single Atomic MultiLog with 16 pre-defined filters. Under Confluo's `non_durable` mode, we persist data at 1s intervals.

Atomic Snapshots. We measured per-server write throughput across $1-8$ storage servers with varying periodicity of atomic snapshots, and found the impact of atomic snapshots on write throughput to be insignificant. In fact, even with 1ms snapshot periodicity, per-server throughput degrades by $< 2\%$ as number of servers increase from 1 to 8. This result might be non-intuitive; the reason is that Confluo only blocks updates to the globalReadTail during the snapshot operation — bulk of the writes including those to StreamLog, IndexLogs and FilterLogs can still proceed, with entire set of pending globalReadTail updates going through at once when the snapshot operation completes. The tradeoff is higher latency because of delayed writes — with 1ms snapshot periodicity across 8 servers, the median and 99%ile latency for Confluo writes are 171μs and 779μs, respectively.

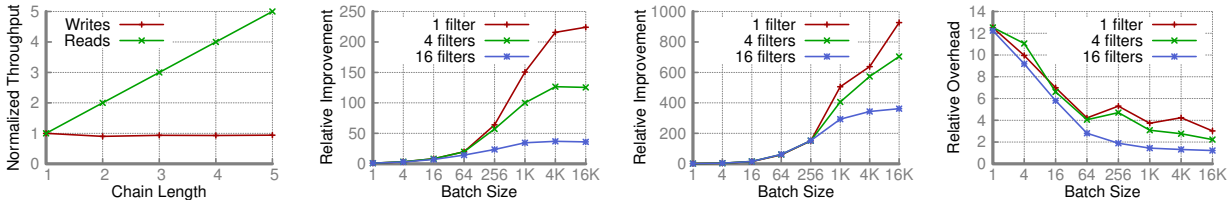
Chain Replication. Figure 9(a) shows Confluo throughput for read and write queries with chain replication. Confluo chain replication protocol (§4.2) ensures that with increase in chain length, write throughput remains close to single server throughput, albeit with increased write latency (increases by $\sim 3.4\times$ for a 5 node chain).

[4] A 70μs latency over 1ms period may result in as high as 7% CPU utilization; we believe the discrepancy is because of the reporting frequency for CPU utilization metrics from the OS.



(a) Single-core write throughput (b) Writes with pre-defined filters only (c) Writes with attribute indexes only (d) Single-core utilization with triggers

Figure 8: (a) Confluo's write throughput (or, data ingestion rate) degrades gracefully on adding attribute indexes and pre-defined filters. The peak throughput scales well with the number of cores, even as the number of (b) pre-defined filters and (c) indexes are increased. (d) Confluo evaluates 1000s of triggers with $< 4\%$ CPU utilization and average latency of $< 70\mu$s at 1ms intervals.



(a) chain replication throughput (b) `non_durable` write throughput (c) `durable` write throughput (d) `durable` vs. `non_durable` writes

Figure 9: (a) Lengthening the replica chain results in negligible write throughput degradation and linear increase in read throughput relative to a single replica (b, c) Batching significantly improves write throughput by mitigating network and SSD write overheads; (d) Durability reduces write throughput by $\sim 3-5\times$ with sufficiently large batches.

As discussed in §4.2, with longer replica chains, Confluo is able to exploit the added servers in the chain to serve read queries resulting in near linear growth in read throughput (Figure 9(a)) without any increase in latency.

Impact of batching. Figure 9(b) shows that write throughput improves significantly for `non_durable` writes on increasing the record batch sizes. This is because small batch sizes observe low throughput due to network overheads, while larger batches amortize the network cost across several records. The improvements on increasing batch sizes are even more significant for `durable` writes (Figure 9(c)), since batching now not only mitigates network overheads, but also SSD write overheads.

Impact of durability. Figure 9(d) compares the relative overheads of `durable` writes over `non_durable` writes. With smaller batches, the overhead for `durable` writes is higher since small batches exhibit poor SSD write throughput. With sufficiently large batches, `durable` write throughput is only $3-5\times$ lower than `non_durable`, since network overheads mask the overhead of persisting `durable` writes for large batch sizes.

6 Confluo Applications

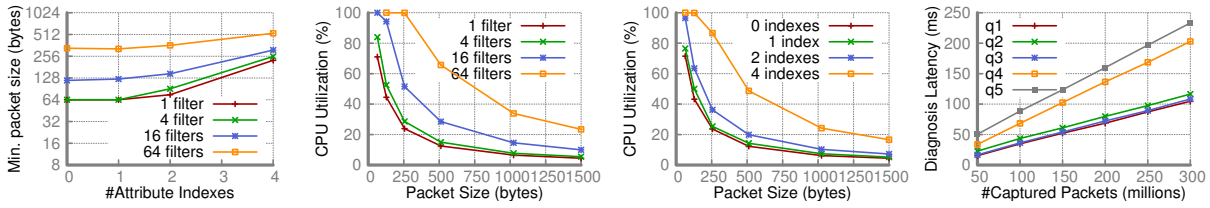
In this section, we describe the design and implementation for two Confluo applications: a network monitoring and diagnosis tool (§6.1) and a time-series database (§6.2). We evaluate each of these applications against state-of-the art approaches in their respective domains.

6.1 Network Monitoring & Diagnosis Tool

Network monitoring and diagnosis is an increasingly challenging task for network operators. Integrating these functionalities within the network stack at the end-hosts allows efficiently using end-host programmability and resources [2, 1, 4, 40] with minimal overheads (end-host stack processes the incoming packets anyways). However, achieving this requires tools that support highly concurrent per-packet capture at line rate, support for online queries (for monitoring purposes), and offline queries (for diagnosis purposes). Confluo's interface is a natural fit for building such a tool — flows, packet headers and header fields at an end-host are equivalent to Confluo streams, records and attributes.

Implementation. We extend Confluo architecture with additional modules to enable capturing and monitoring packets in the hypervisor, where a software switch delivers packets between NICs and VMs. Confluo interfaces with the software switch using a module that writes packet headers to multiple ring buffers in a round-robin manner, which are then picked up by Confluo's writer module and written to the Atomic MultiLog (see Figure 2). We uses DPDK [41] to bypass the kernel stack, and Open vSwitch [42] to implement the module.

Confluo's online and offline modules (§3.4) enable network monitoring and diagnosis functionality. In addition, Confluo's atomic snapshot functionality (§4.1) enables diagnosing events that require headers captured across multiple end-hosts (*e.g.*, if the red and the blue flow in Figure 1 were destined to different end-hosts).



(a) Min. packet size at 10Gbps (b) CPU% at 10Gbps (c) CPU% at 10Gbps (d) Diagnostic Query Latency

Figure 10: (a) The smallest packet Confluo can handle gracefully degrades with #indexes and #filters (b, c) At line rate, Confluo can handle as small as 512B packets with 64 filters, and 256B packets with 4 indexes on a single core with low utilization; with 64B packets, Confluo can handle 20% of line rate with 64 filters and 40% with 4 indexes (d) Diagnostic query latency increases linearly with #captured packets in Confluo, due to larger cardinalities for intermediate results. The filters use the following templates (varying value of A, B, IP, and port for various filters): `(q1)` packets from VM `A` to VM `B`; `(q2)` packets to VM `A`; `(q3)` packets from VM `A` on destination port `port`; `(q4)` packets between $(\texttt{IP}_1, \texttt{prt}_1)$ and $(\texttt{IP}_2, \texttt{prt}_2)$; and `(q5)` packets to or from VM `A`

To the best of our knowledge, none of the existing systems [2, 1, 40, 4] provide such a functionality.

Experimental Setup. Our experiments were conducted on two servers (12-core 2.30GHz Xeon CPUs, 252GB RAM) connected by a 10Gbps link. Synthetic network traffic was generated using DPDK's `pktgen` tool [43] at line rate, and comprised TCP packets with 64 byte headers and arbitrary payloads, IPs drawn from a /24 prefix and ports drawn from 10 common application port values. We use up to 5 attribute indexes (one for each of 4-tuple and packet timestamp), up to 64 filters (defined in Figure 10) and a constant 1000 triggers. Each index, filter and trigger is evaluated on a per-packet granularity.

Packet Capture at 10Gbps. Given fixed link bandwidth of 10Gbps, we can measure the packet ingestion rate Confluo can sustain by measuring the smallest packet size for which Confluo can ingest packets at line rate. Figure 10(a) shows that with small to moderate number of indexes and filters, Confluo can sustain line rate with the smallest packet size (64B). Real-world workloads [44] show that average packet size in datacenter networks is around 850B. Confluo is able to ingest such workloads with each of 5 indexes, 64 filters and 1000 triggers evaluated upon each packet arrival. Confluo can handle workloads with smaller average packet sizes using just one extra core (see Appendix A.4.1).

CPU Utilization at 10Gbps. Figure 10(b) and Figure 10(c) provide more insights on Confluo's performance. For smaller packet sizes along with 4 indexes and 64 filters, CPU becomes a bottleneck; however, CPU utilization drops dramatically with fewer filters or indexes.

Diagnosis Latency. We evaluate Confluo's diagnostic query performance using five queries (`q1` to `q5` outlined in Figure 10). Since these queries combine results from different Confluo IndexLogs, query latency depends on intermediate result cardinalities. Consequently, the query latency increases linearly with the number of captured packets, since cardinalities of intermediate results also grow linearly with the latter. As such, Confluo is able to perform complex diagnostic queries on-the-fly with sub-second latencies on 100s of millions of packets.

6.2 Time-series Database

We now describe extensions to Confluo's interface (Table 1) to capture time-series data and support operations similar to BTrDB [5] on the captured data.

Implementation. Time-series data comprises of a stream of records, each of which is a (timestamp, value) pair. Confluo maintains an IndexLog on both the timestamp and the value attribute to support queries on time windows as well as more general diagnostic queries. Confluo supports efficient aggregate queries using a modification in AggregateLog— each node of the k-ary tree uses a log to store aggregates (Table 2, for any user-defined `aFunction`) for records in the sub-tree below the node.

StreamLog offsets implicitly form versions for the database — each offset corresponds to a new version of the database and includes all records before that offset. Confluo also permits users to compute the difference between two versions of the database: since database versions map to StreamLog offsets, Confluo fetches all records that lie between the StreamLog offsets corresponding to the two versions.

Compared Systems and Setup. We evaluate Confluo against BTrDB [5], CorfuDB [45] and TimescaleDB [8] on c4.8xlarge instances with 18 CPU cores and 60GB RAM. We used the Open μPMU Dataset [46], a real-world trace of voltage, current and phase readings collected from LBNL's power-grid over a 3-month period. We create a separate Atomic MultiLog for each type of reading (voltage, current or phase). We run single server benchmarks with 500 million records to highlight the per-server performance of these systems[5]. Requests are

[5]Each of these systems use a different mechanism for queries across servers, making it hard to gain insights on performance.

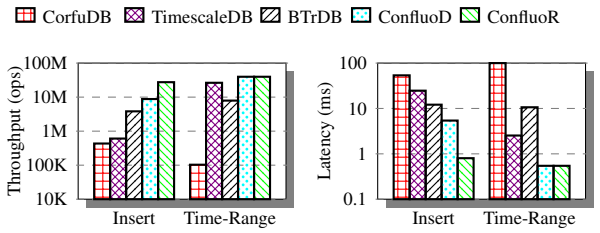


Figure 11: ConfluoD and ConfluoR measure performance for `durable` and `non_durable` modes respectively. ConfluoD achieves 2-20× higher throughput, 2-10× lower latency for inserts, and 1.5-5× higher throughput, 5-20× lower latency for time-range queries than compared systems.

issued as continuous streams with 8K record batches[6].

Results. Figure 11 shows that systems like CorfuDB and TimescaleDB achieve over 10× lower performance than BTrDB and Confluo. We emphasize that this is not a shortcoming: CorfuDB and TimescaleDB support stronger (transactional) semantics than BTrDB and Confluo. Thus, depending on desired semantics, either class of systems may be useful for an underlying application.

BTrDB stores its data directly in a tree; thus, durability requires flushing all modified nodes to SSD for each write. Confluo, on the other hand, only has to persist the data written to the StreamLog thus achieving 2× higher write throughput. Moreover, using `non_durable` writes, Confluo is able to achieve close to 27 million ops due to its cheap versioning and lock-free concurrency control. For time-range queries, almost all systems observe similar throughput since queries are served via in-memory indexes[7]. Insertion and query latency trends (Figure 11(b)) are similar to throughput trends across different systems.

7 Related Work

Given that log-based systems have been around for several decades, it would be impractical to attempt an exhaustive comparison. Instead, we focus on recent works most relevant to Confluo's target workloads.

Atomic MultiLog. There has been a lot of work on the design of efficient, concurrent logs [19, 20, 21, 47, 22, 48]. While these approaches focus on simple atomic operations on a *single* log, Confluo combines a *collection of logs* in the Atomic MultiLog to support atomic filters, aggregates and triggers over multiple data streams. In fact, Confluo relaxes atomicity guarantees for its individual logs, guaranteeing atomicity only for end-to-end MultiLog operations to enable higher concurrency.

Corfu [13], Tango [14] and vCorfu [15] are shared log systems that employ state machine replication to provide the abstraction of replicated, in-memory data structures. These systems use a centralized sequencer to order all writes to the system to enable strong transactional semantics. Unfortunately, this upper-bounds system write throughput by the sequencer throughput (∼ 500KOps [14]), while Confluo's target workloads can generate millions of writes/second per server (§1). Confluo explores a different point in this tradeoff space, supporting performant atomic operations, but without the transactional semantics afforded by shared log systems.

Network Monitoring and Diagnosis Tools. End-hosts have both abundance of resources and desired programmability to efficiently implement monitoring and diagnosis functionalities [1, 2, 4, 40, 3]. However, existing tools for end-host based network monitoring and diagnosis either do not achieve line rate [2, 3, 40], do not support diagnosis queries [1], or focus on diagnosing special class of network problems [4]. Confluo complements these systems, in that, it can be used as a tightly integrated component of the network stack to enhance these systems either in terms of performance [2, 3, 40] or in terms of expressivity [4, 1].

Time-series Databases. A number of recent storage systems cater specifically to time-series data [5, 49, 50, 9, 51, 52, 8]. State-of-the-art transactional systems like TimescaleDB [8] provide stronger semantics, but compromise on write throughput. Non-transactional systems like BTrDB [5] relax their semantics to support only atomic operations, but are able to achieve much higher write rates. Confluo takes the latter approach and improves on BTrDB performance using lock-free operations on Atomic MultiLogs, while supporting richer functionality via filters, aggregates and triggers.

8 Conclusion

We have presented Confluo, a system that achieves three properties useful for real-time monitoring and analysis of data streams: (1) high-throughput concurrent writes of millions of data points from multiple data streams; (2) online queries at millisecond timescale; and (3) ad-hoc queries using minimal CPU resources. Confluo achieves this via a new data structure called Atomic MultiLog, that supports efficiently updating a collection of lock-free concurrent logs as a single atomic operation. We have built two applications on top of Confluo — a network monitoring and diagnosis tool and a time-series database — and showed that Confluo can outperform state-of-the-art systems for these applications.

[6] We use large batch sizes for all systems to mitigate network overheads associated with sending one record at a time.

[7] We observed unexpectedly low performance for time-range queries on CorfuDB. We have been working with the authors to improve the results and report the currently best achievable numbers. We attribute our observed performance to the following: (1) the open-source CorfuDB code-base [45] does not support remote views, which is a source of significant performance benefits reported in [15]; (2) CorfuDB's SMRMap currently uses full-scans to support filter operations.

References

[1] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Trumpet: Timely and Precise Triggers in Data Centers," in *SIGCOMM*, 2016.

[2] P. Tammana, R. Agarwal, and M. Lee, "Simplifying Datacenter Network Debugging with PathDump," in *OSDI*, 2016.

[3] "Improving Network Monitoring and Management with Programmable Data Planes." http://p4.org/p4/inband-network-telemetry/.

[4] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive Realtime Datacenter Fault Detection and Localization," in *NSDI*, 2017.

[5] M. P. Andersen and D. E. Culler, "Btrdb: Optimizing storage system design for timeseries processing," in *FAST*, 2016.

[6] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, "Bolt: Data management for connected homes," in *NSDI*, 2014.

[7] H. Cui, K. Keeton, I. Roy, K. Viswanathan, and G. R. Ganger, "Using data transformations for low-latency time series analysis," in *SoCC*, 2015.

[8] "TimescaleDB: SQL made scalable for time-series data." https://www.timescale.com/papers/timescaledb.pdf.

[9] L. Abraham, J. Allen, O. Barykin, V. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed, "Scuba: Diving into data at facebook," *Proceedings of the VLDB Endowment*, 2013.

[10] "Google Stackdriver." https://cloud.google.com/stackdriver/.

[11] "Amazon CloudWatch." https://aws.amazon.com/cloudwatch/.

[12] "Ganglia Monitoring System." http://ganglia.info.

[13] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "CORFU: A Shared Log Design for Flash Clusters," in *NSDI*, 2012.

[14] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed Data Structures over a Shared Log," in *SOSP*, 2013.

[15] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi, "vCorfu: A Cloud-Scale Object Store on a Shared Log," in *NSDI*, 2017.

[16] "Apache Kafka." https://kafka.apache.org.

[17] "Amazon Kinesis." https://aws.amazon.com/kinesis/.

[18] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-Level Telemetry in Large Datacenter Networks," in *SIGCOMM*, 2015.

[19] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *EuroSys*, 2015.

[20] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *TOPLAS*, 1990.

[21] "A Fast Lock-Free Queue for C++." http://moodycamel.com/blog/2013/a-fast-lock-free-queue-for-c++.

[22] P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *SPAA*, 2001.

[23] P. E. Black, "perfect k-ary tree." https://www.nist.gov/dads/HTML/perfectKaryTree.html.

[24] "Disjunctive normal form." https://en.wikipedia.org/wiki/Disjunctive_normal_form.

[25] "SQLServer: Distributed Transactions (Database Engine)." https://technet.microsoft.com/en-us/library/ms191440(v=sql.105).aspx.

[26] "Oracle: Distributed Transactions Concepts." https://docs.oracle.com/cd/B10501_01/server.920/a96521/ds_txns.htm.

[27] "Postgres: eXtensible Transaction Manager." https://wiki.postgresql.org/wiki/DTM.

[28] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Transactions on Computer Systems*, 1985.

[29] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic Snapshots of Shared Memory," *Journal of the ACM (JACM)*, 1993.

[30] H. Attiya and O. Rachman, "Atomic Snapshots in O (N Log N) Operations," *SIAM Journal on Computing*, 1998.

[31] H. Attiya, M. Herlihy, and O. Rachman, "Atomic snapshots using lattice agreement," *Distributed Computing*, 1995.

[32] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *OSDI*, 2004.

[33] J. Terrace and M. J. Freedman, "Object storage on craq: High-throughput chain replication for read-mostly workloads," in *USENIX ATC*, 2009.

[34] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling Queries on Compressed Data," in *NSDI*, 2015.

[35] "Configuring compression in Cassandra." https://docs.datastax.com/en/cassandra/2.0/cassandra/operations/ops_config_compress_t.html.

[36] "RocksDB Tuning Guide." https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[37] "Memtables in Cassandra." https://wiki.apache.org/cassandra/MemtableSSTable.

[38] "Configuring compaction in Cassandra." https://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_configure_compaction_t.html.

[39] "SSTable and Log Structured Storage: LevelDB." https://www.igvita.com/2012/02/06/sstable-and-log-structured-storage-leveldb.

[40] H. Chen, N. Foster, J. Silverman, M. Whittaker, B. Zhang, and R. Zhang, "Felix: Implementing Traffic Measurement on End Hosts Using Program Analysis," in *SOSR*, 2016.

[41] "Intel Data Plane Development Kit (DPDK)." http://dpdk.org.

[42] "Open vSwitch (OVS)." http://openvswitch.org.

[43] "The Pktgen Application." https://pktgen.readthedocs.io/en/latest/.

[44] T. A. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," in *Computer Communication Review*, 2009.

[45] "CorfuDB." https://github.com/CorfuDB/CorfuDB.

[46] E. M. Stewart, A. Liao, and C. Roberts, "Open μpmu: A real world reference distribution micro-phasor measurement unit data set for research and application development," 2016.

[47] "Lock-Free Programming." https://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf.

[48] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box concurrent data structures for numa architectures," in *ASPLOS*, 2017.

[49] "OpenTSDB: The Scalable Time Series Database." http://opentsdb.net.

[50] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proceedings of the VLDB Endowment*, 2015.

[51] "InfluxDB." https://www.influxdata.com.

[52] M. Buevich, A. Wright, R. Sargent, and A. Rowe, "Respawn: A distributed multi-resolution time-series datastore," in *RTSS*, 2013.

[53] P. Tammana, R. Agarwal, and M. Lee, "CherryPick: Tracing Packet Trajectory in Software-Defined Datacenter Networks," in *SOSR*, 2015.

[54] P. Prakash, A. Dixit, Y. C. Hu, and R. Kompella, "The tcp outcast problem: Exposing unfairness in data center networks," in *NSDI*, 2012.

A Appendix

A.1 Atomic Concurrent Log

We show that it is possible to support concurrent read and append operations on a log atomically using only two markers: a readTail and a writeTail. The writeTail marks the actual end of the log, while the readTail lags behind the writeTail and marks the end of completed append operations. The region between readTail and writeTail correspond to ongoing appends. The update mechanism for the two tails ensures *atomicity* and *total order* across concurrent log operations, as we show below.

Every new append operation first atomically increments the writeTail by the number of bytes it intends to write using the `FetchAndAdd` instruction, and in the process obtains a unique offset within the log. Note that this atomic operation resolves all append-append conflicts, since it allows concurrent append operations to obtain exclusive write access to the region in concurrent log marked by the writeTail values before and after the atomic increment, without acquiring locks. Each append operation ends by atomically updating readTail, making the data it wrote available to subsequent read operations.

The readTail acts as a guard for "read-safe" data in the concurrent log — read operations can only access data before the readTail in the log, ensuring safety. Since read operations only atomically access the readTail without modifying it, they do not contend with concurrent append operations. The readTail also imposes a *total order* among append operations based on their start offsets in the concurrent log: the log only permits an append operation to update the readTail value after all append operations with smaller offsets in the log have updated the readTail, via repeated `CompareAndSwap` instruction attempts. This ensures that there are no "holes" in the read-safe region before the readTail, and allows read operations to ensure correctness via a simple check on the readTail value.

We outline the algorithms for `append` and `read` operations on a concurrent concurrent log in Algorithm 2 and Algorithm 3, respectively.

Algorithm 2 append operation in concurrent log

```
1: procedure append(data, len)        ▷ Append len bytes of data to log.
2:     off ← FetchAndAdd(writeTail, len)
3:     Write len bytes of data starting at offset off
4:     do
5:         success ← CompareAndSwap(readTail, off, off+len)
6:     while ¬ success
7: end procedure
```

Proof of Correctness. We show that concurrent concurrent log operations are *linearizable* to the sequential specification of an abstract log [20]. We start by defining the sequential specification, i.e., the description of a *sequential* abstract log implementation.

Algorithm 3 `read` operation in concurrent log

```
1: procedure read(off, len)           ▷ Read len bytes at offset off in log.
2:     readBoundary ← AtomicLoad(readTail)
3:     if off ≥ readBoundary then
4:         return NULL
5:     end if
6:     readLen ← min(len, readBoundary−off)
7:     data ← Read readLen bytes at offset off
8:     return data
9: end procedure
```

Definition A.1 *A log is an object that supports two types of operations: appends and reads. The state of a log is a continuous block of data $\mathscr{D}$, and is initially empty. Append and read operations induce the following transitions on $\mathscr{D}$:*

- **T1:** *`append(data, len)` modifies $\mathscr{D}$ to $\mathscr{D}\cdot$***data**, *where "$\cdot$" is simple concatenation;*
- **T2:** *`read(off, len)`, if $\mathscr{D}$ is not empty and contains offset* **off**, *returns up to* **`len`** *bytes of data at offset `off` in $\mathscr{D}$, and returns empty otherwise. In both cases, $\mathscr{D}$ remains unchanged.*

Concurrent log operations are linearizable since each operation executed by a concurrent processes takes effect instantaneously at a point between its invocation and response [20]. We identify these *linearization points* for each concurrent log operation in Lemma A.1.

Lemma A.1 *The following are valid linearization points for concurrent log operations:*

- *Every `append` operation takes effect exactly when the atomic `CompareAndSwap` operation succeeds in updating the readTail (Line 5 in Algorithm 2).*
- *Every `read` operation takes effect when at `AtomicLoad` instruction on the readTail (Line 2 in Algorithm 3).*

Proof We identify a concurrent log's state $\mathscr{D}$ as the continuous block of data bounded by the readTail. Therefore, log state is modified only via readTail updates.

Note that each `append(data, len)` operation updates the readTail by `len` amount exactly when the atomic `CompareAndSwap` instruction succeeds in Line 5 of Algorithm 2, modifying the state of the concurrent log to $\mathscr{D}\cdot$`data`. Since this is in line with transition **T1** in Definition A.1, this is a valid linearization point for the `append` operation.

Similarly, the `read(off, len)` operation obtains an atomic snapshot of the concurrent log state $\mathscr{D}$ when it

obtains the readTail value using the `AtomicLoad` instruction in Line 2 of Algorithm 3; the subsequent execution of the read operation leaves the state $\mathscr{D}$ unmodified, and is in line with transition **T2** of Definition A.1, making the `AtomicLoad` instruction a valid linearization point for the `read` operation. ■

Theorem A.2 *Concurrent log described above is a correct linearizable implementation of a log object (Definition A.1).*

Proof Follows from Lemma A.1. ■

Algorithm 4 `write`(r)
Write a new record r to the Atomic MultiLog.

```
1:  rLen ← sizeof(r)
2:  rOff ← FetchAndAdd(globalWriteTail, rLen)
3:  Append r at the end of StreamLog at offset rOff.
4:  for each attribute in record do
5:      if attribute is indexed then
6:          Add (attribute, rOff) to its IndexLog
7:      end if
8:  end for
9:  for each fExpression in Atomic MultiLog do
10:     if r satisfies fExpression then
11:         Add (currentTimeMs, rOff) to its FilterLog if defined
12:         Update corresponding AggregateLog if defined
13:     end if
14: end for
15: do
16:     ok ← CompareAndSwap(globalReadTail, rOff, rOff + rLen)
17: while ¬ok
```

Algorithm 5 `get_aggregate`(`aggId`, `tLo`, `tHi`)
Get aggregate value for `aggId` in time range (`tLo`, `tHi`).

```
1:  version ← AtomicRead(globalReadTail)
2:  aggLog ← AggregateLog corresponding to id
3:  for each Time bucket t in aggLog ∈ [tLo, tHi] do
4:      for each Confluo Writer thread T do
5:          agg_{t,T} ← Most recent aggregate for thread T in time-bucket t with
            version < version
6:      end for
7:  end for
8:  agg ← Aggregate agg_{t,T} ∀t, T
9:  return agg
```

Algorithm 6 `query`(`fExpression`, `tLo`, `tHi`)
Get iterator over records that match `fExpression` in time range (`tLo`, `tHi`).

```
1:  maxOff ← AtomicRead(globalReadTail)
2:  if fExpression has associated FilterLog then
3:      fLog ← FilterLog associated with fExpression
4:      return Iterator over records with offset < maxOff in fLog, for time-
        buckets ∈ [tLo, tHi]
5:  else
6:      return Iterator over records with offset < maxOff obtained by combin-
        ing IndexLog range-query results (§3.4)
7:  end if
8:  return it
```

A.2 Atomic MultiLog

Atomic MultiLog Operations. The execution of Confluo`write`, `get_aggregate` and `query` operations are outlined in Algorithms 4, 5 and 6, respectively. As explained in §3.3.2, `write` operations spanning multiple concurrent logs begin by incrementing the globalWriteTail and end by updating the globalReadTail. Similarly, `get_aggregate` and `query` operations begin by atomically reading the globalReadTail, and only access records with offsets or aggregates with versions smaller than it. Intuitively, this results in `write` operations updating the StreamLog, IndexLogs, FilterLogs and AggregateLogs as a single atomic unit, since their effect is only made visible when the globalReadTail is updated. Similarly, `get_aggregate` and `query` operations are rendered atomic since reading the globalReadTail provides an atomic snapshot of the Atomic MultiLog.

Proof of Correctness. We now show that Atomic MultiLog is linearizable to the sequential specification of an abstract record store, as defined below.

Definition A.2 *A record store is a collection of records that supports write, read and query operations on the records. The state of an record store is represented by the tuple ($\mathscr{R}$, $\mathscr{P}$), where $\mathscr{R}$ is a collection of records {r_1, r_2, ..., r_n}, and $\mathscr{P}$ is the set of references to these records required for supporting the queries efficiently. The record store operations induce the following state transitions:*

- **T1** `write(r)` *modifies ($\mathscr{R}$, $\mathscr{P}$) to ($\mathscr{R} \cup \{r\}$, $\mathscr{P} \cup \mathscr{P}_o$), where $\mathscr{P}_o$ are references to record r that need to be added to the record store based on application defined semantics* (e.g., *indexes or filters). The operation returns the identifier* id *to r.*
- **T2** `get_aggregate(id, t1, t2)` *returns the precomputed aggregate corresponding to the aggregateId id in time-range (t_1, t_2) if such an aggregate exists, otherwise returns empty; in both cases, the state ($\mathscr{R}$, $\mathscr{P}$) remains unchanged.*
- **T3** `query(e, t1, t2)` *returns all records that satisfy expression e in $\mathscr{R}$ in the time range (t_1, t_2), without modifying the state ($\mathscr{R}$, $\mathscr{P}$).*

As before, we show that Atomic MultiLog operations are linearizable to the specification in Definition A.2 by identifying linearization points for `write`, `get_aggregate` and `query` operations, and mapping them to state transitions in Definition A.2.

Lemma A.3 *The following are valid linearization points for our record store implementation's operations:*

- *Every* `write` *operation takes effect exactly when the the atomic* `CompareAndSwap` *operation succeeds in updating the globalReadTail (Line 16 of Algorithm 4).*

- *Each* `get_aggregate` *and* `query` *operation takes effect at the* `AtomicLoad` *instruction on the globalReadTail (Line 1 of Algorithm 5, Line 1 of Algorithm 6).*

Proof The proof structure is quite similar to the proof of Lemma A.1. Atomic MultiLog state is identified as the tuple ($\mathscr{R}$, $\mathscr{P}$), where $\mathscr{R}$ is the collection of records whose StreamLog offsets are smaller than the globalReadTail, and $\mathscr{P}$ is the collection of all StreamLog offsets across different IndexLogs, FilterLogs and AggregateLogs smaller than the globalReadTail. Note that the record store state is only modified through updates to the globalReadTail.

Each `write(`r`)` increments the globalReadTail when the atomic `CompareAndSwap` operation is successful in Line 16 of Algorithm 4, and the record r and its references $\mathscr{P}_r$ become visible to Confluo operations. Confluo state is effectively modified from ($\mathscr{R}$, $\mathscr{P}$) to ($\mathscr{R} \cup \{r\}$, $\mathscr{P} \cup \mathscr{P}_r$) at this point, in keeping with transition **T1** of Definition A.2.

Similarly, the `get_aggregate(`id`,` t_1`,` t_2`)` and `filter(`e`,` t_1`,` t_2`)` operations obtain a snapshot of the record store state when they atomically read the globalReadTail (Line 1 of Algorithm 5, Line 1 of Algorithm 6); the subsequent executions of the `get_aggregate` and `query` operations do not modify Confluo state, only access records and their StreamLog offsets that lie within the globalReadTail and are in line with transitions **T2** and **T3** of Definition A.2 respectively. This renders the `AtomicLoad` on the globalReadTail a valid linearization point for both operations.

Theorem A.4 *Atomic MultiLog is a correct linearizable implementation of a sequential record store (Definition A.2).*

Proof Follows from Lemma A.3. ■

A.3 Comparison Against Apache Kafka

Distributed messaging systems such as Kafka [16] and Kinesis [17] employ partitioned logs to serve streams of messages using a publish-subscribe model. Confluo builds on this model to additionally provide lock-free resolution of read-write and write-write contentions for significantly improved publish/subscribe throughput.

Implementation. Confluo employs Kafka's interface and data model — messages are published to or subscribed from "topics", which are logically streams of messages. The system maintains a collection of topics, where messages for each topic are stored across a user-specified number of ConfluoAtomic MultiLogs.

Confluo publishers write messages in batches to MultiLogs of a particular topic, and subscribers asynchronously pull batches of messages from them. Similar to Kafka, each subscriber keeps track of the offset for its last read message in the MultiLog, incrementing it as it consumes more messages. The key benefit of using Confluo for storing messages is the freedom from read-write contentions, and lock-free resolution of write-write contentions. Additionally, Confluo provides an efficient means to obtain the snapshot of an entire topic, unlike Kafka.

Compared Systems and Setup. We compare performance for Confluo against Apache Kafka. Since both systems are identical in terms of scaling read and write performance via log partitions, we ran our experiments on 2 r3.8xlarge instances (configured as server and client), using a single topic with one log partition for both systems. Our experiments used messages with an average size of 64B, while concurrent subscribers in both systems are kept in independent subscriber groups, i.e., perform uncoordinated reads on the partition. We mount Kafka's storage on a sufficiently sized RAM disk, ensuring both systems operate completely in memory.

Results. Since Kafka employs locks to synchronize concurrent appends, publisher write throughput suffers due to write-write contentions (Figure 12 (left)). Confluo employs lock-free resolution for these conflicts to achieve high write throughput. Larger batches (16K messages) alleviate locking overheads in Kafka to some extent, while Confluo approaches network saturation at 16K message batches with over 4 publishers. Since reads occur without contention in both systems, read throughput scales linearly with multiple subscribers (Figure 12 (center)). Confluo achieves higher absolute read throughput, presumably due to system overheads in Kafka and not because of a fundamental design difference. As before, read throughput for Confluo saturates at 4 subscribers and 16K message batches due to network saturation. Confluo latency increase is sub-linear with the batch size for both reads and writes, while Kafka observes super-linear increase in latency, particularly for writes (Figure 12 (right)). Again, we suspect this effect is likely due to implementation overheads in Kafka.

A.4 Network Monitoring and Diagnosis

We now present additional results for network monitoring and diagnosis using Confluo.

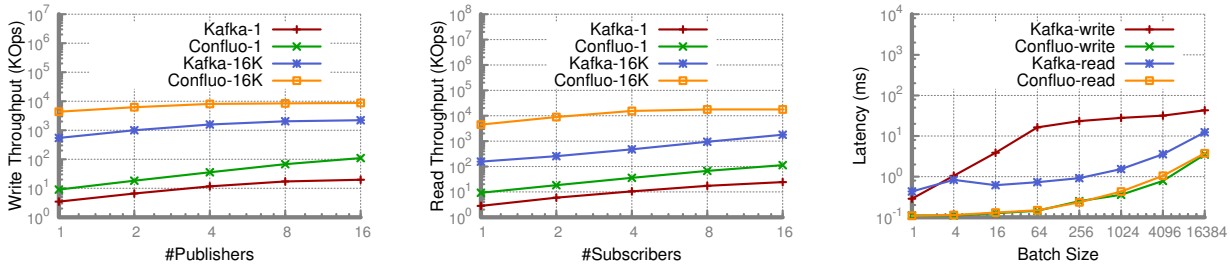


Figure 12: ConfluoStream observes close to linear write throughput scaling with #publishers as opposed to Kafka's sub-linear scaling, while both systems observe close to linear read throughput scaling with #subscribers. ConfluoStream observes sub-linear increase in read and write latency with batch sizes as opposed to Kafka's linear increase (both axes are in log-scale).

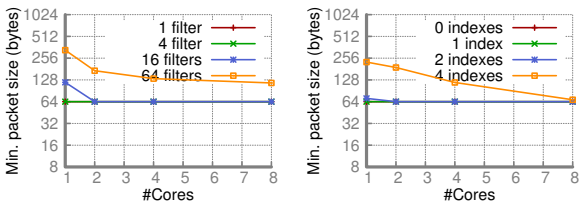


Figure 13: Minimum packet size Confluo can handle at 10Gbps Using multiple cores, NetPlay can expand its feasibility region for both (a) indexes, and (b) pre-defined filters.

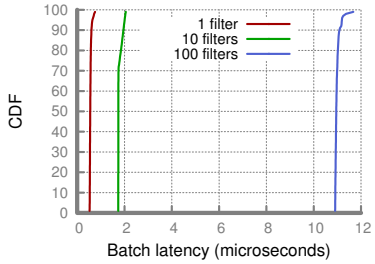


Figure 14: Confluo latency for path conformance checks. Confluo can perform 100s of checks while capturing packet headers, incurring per-packet latency of 350ns.

A.4.1 Confluo Multi-Core Performance

We saw in §6.1 that Confluo can handle as small as 64B packets with a moderate number of indexes and filters, on a single core. Confluo can handle much larger number of indexes and filters using multiple cores as shown in Figures 13, using the same setup as §6.1. This is in line with the recent server architecture trend of equipping large number of CPU cores at each end-host.

A.4.2 Debugging Network Issues

We use Confluo to detect and debug a variety of network issues in modern data center networks. We debugging these network issues on servers with 2×12-core 2.30GHz Xeon CPUs and 252GB RAM with 10Gbps access links, and employ techniques from CherryPick [53] to trace switches that a packet traverses in its header.

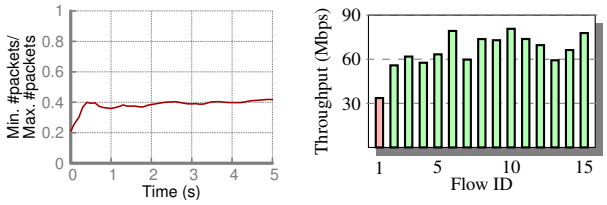
Path Conformance. We demonstrate Confluo's ability to quickly monitor and debug path conformance violations by randomly routing some packets *within a flow* via a particular switch S, violating network policy. Confluo is configured with a single AggregateLog that tracks the number of packets from the specified flow that pass through switch S. A companion trigger generates an alert every time the packet count is non-zero. Confluo evaluates the trigger at 10 ms intervals, alerting the presence of path non-conformant packets within milliseconds of receiving it at the end-host.

Figure 14 shows the latency for processing a batch of 32 packets in Confluo with varying number of path conformance checks. We note that while a single check incurs average batch latency of 1μs (30 ns per packet), 100 checks incur 11μs (350 ns per packet), indicating sub-linear increase in latency with the number of checks. As such, Confluo is able to perform *per-packet* path conformance checks with minimal overheads.

TCP Outcast. Confluo can both detect *and* diagnose traffic patterns corresponding to the TCP outcast problem [54]. In our experiment, we use a setup where 15 TCP flows with different sources and the same destination compete for a single output port at the final-hop switch. While one flow traverses a 1-hop path, two of them traverse a 3-hop path, while the remaining 12 traverse a 5-hop path. All links in the setup have 1Gbps bandwidth.

The destination end-host installs a AggregateLog at Confluo which tracks retransmitted TCP packet count, along with a trigger that checks if there were more than 10 retransmitted packets in the last 10ms interval. When the trigger is satisfied, Confluo issues diagnostic queries at 100ms intervals to obtain packet count for each flow in that window, and compute cumulatively (1) ratio of the smallest to largest packet counts across all flows, and (2) the individual flow throughputs (Figure 15).

Owing to the port blackout phenomenon [54], the flow



(a) Packet Ratio (b) Flow Throughput

Figure 15: Diagnosing TCP Outcast. Confluo measures the cumulative ratio of the smallest and largest number of packets received across all flows at 100ms intervals to diagnose outcast. The smallest and largest packet counts correspond to flows with smallest and largest hop-counts respectively, with their ratio stabilizing to 0.4 in 1 s after measurement starts, as shown in (a). Individual flow throughputs at $t = 1$ s are shown in (b).

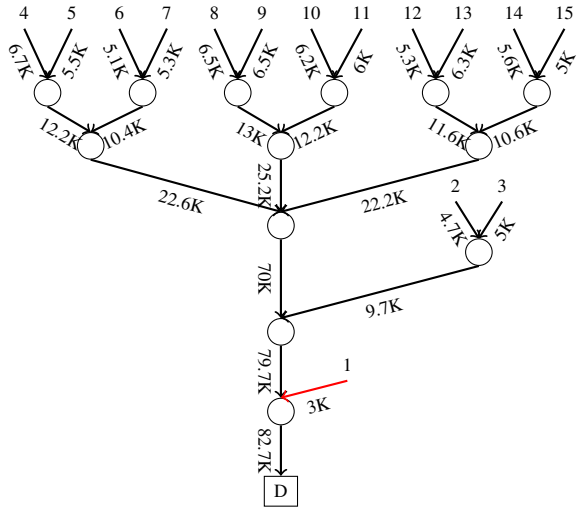
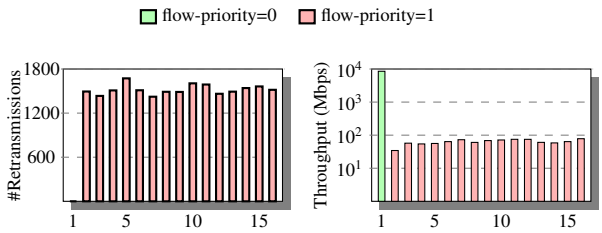


Figure 16: Packet distribution across links in TCP Outcast. Using the trace of switches each packet traverses, Confluo is able to estimate the distribution of packets across network links (numbers along the links) in a 1s window during outcast. $1-15$ represent different flow IDs from Figure 15(b), the circles represent switches, and D corresponds to the destination.



(a) Retransmission Count (b) Flow Throughput

Figure 17: Correlating retransmission count and throughput with flow priority. Confluo measures (a) retransmission count and (b) throughput for every flow to detect losses due to flow priorities. This figure shows a snapshot for these metrics, grouped by flow priorities, at time $t = 1$s after measurement starts. Note that (b) uses log-scale on y-axis.

with the smallest hop-count achieves the lowest throughput, while the flows with larger hop-counts observe higher throughput. The ratio of the packet counts for the two stabilize to 0.4 in about a second after the retransmission trigger is satisfied, as shown in Figure 15(a). Figure 15(b) shows the individual flow throughputs at time $t = 1$s. With the help of the path traces available in the packet headers, Confluo is also able to estimate the number of packets transmitted through each link in the network over a 1 second window, as shown in Figure 16. Executing each of the diagnostic queries incurs an average latency of 8μs, demonstrating Confluo's ability to detect and diagnose TCP outcast with very low resource utilization at the end-host.

Losses due to Flow Priorities. When flows with different priorities compete for bandwidth at an output switch interface, flows with lower priorities can observe severe losses and degradation in throughput. In such scenarios, detecting the issue and scheduling low priority flows on other interfaces can lead to better network utilization. In our experiment, we used a toy setup with 15 low priority flows and 1 high priority flow competing for bandwidth at a 10Gbps access link to a common destination.

Similar to TCP outcast, the destination end-host maintains AggregateLogs to track TCP retransmission counts, but for *every flow*, and an additional trigger to check if the total retransmission count exceeds 10 in the last 10ms. Upon trigger satisfaction, Confluo installs additional AggregateLogs to compute throughput for each flow at 100 ms intervals, and attempts to correlate them with with flow priorities. Figure 17 shows the snapshot of these measurements correlated with the flow priorities at $t = 1$s.

Since a clear correlation is present between flow priorities and throughput, Confluo concludes that flow priorities are responsible for the losses, performs additional diagnosis to determine the common switch at which different flows collide, using switch traces in the packet headers similar to Figure 16 for TCP outcast.