Cluster Computing
https://doi.org/10.1007/s10586-021-03463-5

Using advanced data structures to enable responsive security monitoring

Janet Vorobyeva[1] · Daniel R. Delayo[2] · Michael A. Bender[2] · Martín Farach-Colton[3] · Prashant Pandey[4] · Cynthia A. Phillips[1] · Shikha Singh[5] · Eric D. Thomas[1] · Thomas M. Kroeger[1]

Received: 17 May 2021 / Revised: 10 September 2021 / Accepted: 30 October 2021

Abstract
Write-optimized data structures (WODS), offer the potential to keep up with cyberstream event rates and give sub-second query response for key items like IP addresses. These data structures organize logs as the events are observed. To work in a real-world environment and not fill up the disk, WODS must efficiently expire older events. As the basis for our research into organizing security monitoring data, we implemented a tool, called *Diventi*, to index IP addresses in connection logs using RocksDB (a write-optimized LSM tree). We extended *Diventi* to automatically expire data as part of the data structures' normal operations. We guarantee that *Diventi* always tracks the N most recent events and tracks no more than $N + k$ events for a parameter $k < N$, while ensuring the index is opportunistically pruned. To test *Diventi* at scale in a controlled environment, we used anonymized traces of IP communications collected at SuperComputing 2019. We synthetically extended the 2.4 billion connection events to 100 billion events. We tested *Diventi* vs. Elasticsearch, a common log indexing tool. In our test environment, Elasticsearch saw an ingestion rate of at best 37,000 events/s while *Diventi* sustained ingestion rates greater than 171,000 events/s. Our query response times were as much as 100 times faster, typically answering queries in under 80 ms. Furthermore, we saw no noticeable degradation in *Diventi* from expiration. We have deployed *Diventi* for many months where it has performed well and supported new security analysis capabilities.

Keywords Write-optimized · Security monitoring · Network monitoring · Expiration

Janet Vorobyeva and Daniel R. Delayo have contributed equally to this research.

✉ Thomas M. Kroeger
tmkroeg@sandia.gov

Janet Vorobyeva
jvoroby@sandia.gov

Daniel R. Delayo
ddelayo@cs.stonybrook.edu

Michael A. Bender
bender@cs.stonybrook.edu

Martín Farach-Colton
martin@farach-colton.com

Prashant Pandey
ppandey@lbl.gov

Cynthia A. Phillips
caphill@sandia.gov

Shikha Singh
shikha.singh@williams.edu

Eric D. Thomas
edthoma@sandia.gov

[1] Sandia National Labs, Albuquerque, USA

[2] Stony Brook University, Stony Brook, USA

[3] Rutgers University, New Brunswick, USA

[4] Lawrence Berkeley National Lab, Berkeley, USA

[5] Williams College, Williamstown, USA

Published online: 24 January 2022

1 Introduction

Recent efforts in cybersecurity seek to build real-time analytic tools to respond to threats as they occur. Even basic behavioral analytics, e.g., determining whether the originating IP in an secure shell (SSH) connection has spoken with 5 other hosts (reasonable) or 500 (suspicious), can be a powerful capability, *as long as these analytics can work in real-time*. If we can make these kinds of determinations as events occur, or soon after, we can build security systems that respond automatically to threats, for example, by recording all packet data for suspicious activity.

For behavioral analytics to accurately reflect observed activity, an analytic must consider a long window of sensor data (e.g., months to years). In cybersecurity, sensor data is often summarized network-traffic in the form of Zeek [17] connection logs or netflow/IPFix logs from a router [8, 14], and the volumes of data collected are so high that querying months of data can be time consuming. To respond to threats, we *must* be able to execute these queries quickly, ideally on sub-second timescales.

Security analysts often organize sensor logs by inserting them into a general-purpose log-indexing tool like Elasticsearch or Splunk, which answer queries faster than simply scanning the logs. To keep up with line-rate event streams, these general-purpose tools typically require massive amounts of parallelism (i.e., clustered deployments). In spite of these extensive resources, these tools often answer queries in minutes to hours, practically limiting the questions that are asked and inhibiting timely responses. Developers of analytics often use a custom-built in-RAM index to enable near real-time queries, but these are limited to the size of RAM. They can store only a brief window of recent events, and often quickly lose accuracy as the data grows larger than RAM [3].

In this paper we show how to achieve both sub-second query times and ingestion at line rate for months to years of network events by using WODS to organize data as it arrives. We further extend our WODS to prune older data, enabling sustained operations on a fixed-sized disk, while still keeping up with line rates. Our basis for this work is an open-source tool we created, called *Diventi*. *Diventi* uses a write-optimized key-value store called RocksDB [11, 13] to index the IP addresses of events in network connection logs. RocksDB provides an open source implementation of a log-structured merge tree (LSM tree), a common and well understood WODS.

By selecting IP addresses as an important class of queries that need to be answered quickly, we can create near real-time behavior analytics like the SSH example above. *Diventi* doesn't answer generic queries, but it does answer specific queries like this in a way that could be used to respond to threats. For example, in our SSH analytic mentioned above, an analytic that works in milliseconds could route the traffic for the IP with 500 SSH connections through a slower path that records all traffic for further analysis.

Furthermore, our approach compliments general purpose search tools: *Diventi* provides responders with up to the minute behavioral summaries and an immediate view of a situation as events unfold. Tools like *Diventi* can provide data enrichment, which can significantly improve real-time situation awareness for incident responders. For example, *Diventi* can provide a summary of connection behavior when a specific IP is moused over in a security information and event manager (SIEM). At the same time, the general purpose search tools are necessary for more heavyweight follow-up queries, which can be more effectively targeted with the situational awareness that *Diventi* provides.

While this work focuses on IP addresses, the same approach could provide similar capabilities for other key items, such as DNS names or e-mail addresses. Our point in this work is to show that by thinking about how we organize our security-monitoring data, significant gains in real-time behavioral analytics can be achieved. Tools like this are necessary if we are to move from analytics that simply alert to systems that take actions to respond based on long-term behaviors covering months and years worth of data.

Our initial work using WODS to index IP addresses brought to light one key limitation of the current tools: the need to efficiently expire old information as new data is ingested. While typical WODS commonly have a deletion function, they are costly. In a steady state every insert would require a deletion to ensure the database didn't grow too big and fill up the disk. Yet if a typical deletion is done with each insert we would quickly grind our system to a crawl. This is because WODS typically optimize insertions at the cost of slowing down lookups by 10–100 times, and each deletion could require a lookup of the item to enable deletion. Alternatively, many general purpose systems like Elasticsearch use multiple rolling indices, discarding old data by dropping the oldest index. Unfortunately, as we show later, this approach causes query performance to scale poorly with size.

To enable expiration, we implemented a time-stamp histogram that allows us to loosely track which time-stamp corresponded to the *N*th event. We then created a custom compaction filter within RocksDB to opportunistically purge events older than a given time-stamp. This expiration algorithm guarantees the index always has the N most recent events and takes advantage of the LSM tree's compaction process to opportunistically expire data. The

result of this work is an LSM tree that maintains a rolling window that indexes the N most recent events with some excess k. Our analysis in Sect. 4 shows that we do not expect ingestion rates and query response times to be negatively impacted.

1.1 Real-world and benchmarking

We performed initial tests of *Diventi* at SuperComputing (SC) as part of the Network Research Exhibition (NRE) track. We indexed security events at rates well above 170,000 events/s and responded to queries in milliseconds, without ever approaching I/O or CPU saturation.

While SC was helpful for prototyping and experimentation, to benchmark our systems we needed a consistent and reproducible workload for standardized tests. We collected and anonymized traces of 2.4 billion connection records seen at SC 2019. While a dataset of 2.4 billion connections is rare and unique, it was not large enough for the long-term workloads we needed to test our system. To address this, we extended the SC traces to 100 billion events by looping them and adjusting the time stamp on each loop. While this synthetic workload has limitations in how it represents real-world behavior (see Sect. 5.2), all of the synthetic workloads we examined had limitations and this approach seemed to provide the best balance of realism and efficiency.

Benchmarking *Diventi* with this workload, we found that it performed more than fast enough to support near real-time queries. Typical queries returned in under 80 ms, and even large queries returning over 1 million distinct events completed in 8 s. For comparison, running the same tests on Elasticsearch took almost 100 times longer to answer the same queries. *Diventi* also performed well on ingestion rates, managing 170,000 insertions/s at a database size of 25 billion events, while still managing 115,000 insertions/s when we let the data grow to 100 billion events.

Beyond SC and synthetic workloads, our system has also seen long-term near real-world usage on live networks indexing over 430 billion events and expiring many months' worth of data. From the RESTful interface, analysts were able to easily extend the tool with a basic script that queries months of traffic in seconds. They use this tool to create a near real-time behavioral summary of any given IP. Our code, the anonymized traces from SC, and associated tools have been made publicly available with this publication.

1.2 Contributions

This paper presents three key contributions:

- We present a new approach to organizing security-monitoring data by focusing on near real-time response for key events. We build on recent advances in WODS to maintain a single, strong index able to answer queries across months of data, with results returned in milliseconds. This enables near real-time behavioral analytics for key events, a key capability gap we recognized from our experience in security monitoring.
- We present an efficient method to handle expiration of old data that ensures we always keep the N most recent events indexed and never keep more than $N + k$ events, for user parameter N and parameter $k < N$ that depends on other parameters. This gives a lower bound on data available for queries and an upper bound on space usage. Our analysis showed that we could efficiently integrate culling of older data as part of RocksDB's normal data-structural operations with minimal performance overhead, even when data arrive out of order.
- We collected a uniquely large dataset at SC 2019 consisting of anonymized traces of 2.4 billion network connections. By extending these traces synthetically, we created a workload of 100 billion events that enabled us to model some realistic long-term behaviors for our approach and expiration methods. We are publishing this anonymized dataset alongside the release of this paper.

This paper is organized as follows. Section 2 provides some initial background on write-optimized data structures, and discusses related work. Section 3 presents the design of our IP indexing tool. Section 4 provides some analysis of the expected performance for our expiration algorithm. Section 5 provides details of our testing methodology and our synthetic trace generation. Section 6 presents the results from our laboratory tests. We provide future work and concluding remarks in Sect. 7.

2 Background

In this section we provide some background on write-optimized data structures and security monitoring.

2.1 Security monitoring

Computer security monitoring has focused on collecting and analyzing network events for years. Tools like Zeek [17] and DHS's Einstein program [10] have made great strides in logging high-speed, real-time network interactions. For each conversation between two IP addresses, these flow-logging tools record metadata such as the timestamp, ports used, the number of packets and bytes exchanged in each direction, and basic connection state

flags. This metadata provides an effective way to summarize network-level events.

Flow event logs are frequently fed into tools like Splunk or Elasticsearch to provide a common point of entry for incident responders to query. The scale of this data often requires large clustered deployments of indexers and search heads to both keep up with the rate of incoming data and enable queries suitable for human workflows, on the order of seconds for short timeframes. When analysts need to understand behaviors for longer terms, months or more, such queries can take minutes or hours, depending on their complexity, forcing analysts to switch contexts frequently, reducing their analytical effectiveness. High-latency queries also inhibit automated workflows, such as for real-time response and data enrichment.

To address query delay, some systems either rely on RAM-based data structures or probabilistic structures like Bloom filters [4], but these tools have limited ability to track events and accurately represent behaviors. Berry and Porter [3] showed that pattern-detection efficacy of a state of the art RAM-based structure quickly declines as the data grows beyond the size of the structure. Specifically, when the data set tracked was the size of the structure, the capability detected 66% of reportable events. When the data set was twice and four times the size of the structure, the detection rate dropped to 23% and 0.053%, respectively. These detection errors are false negatives, unknowable to analysts.

In short, we believe there is a need for efficient low-maintenance methods to index security-monitoring events at scales on par with the amount of data currently being collected, and we believe write-optimized data structures offers a valuable tool for tracking this flood of data.

2.2 Write-optimized data structures (WODS)

WODSs [2] balance primary memory and secondary storage and support significantly higher ingestion rates, in practice an improvement of as much as two orders of magnitude compared to traditional indexing data structures such as *B*-trees. Examples of write-optimized data structures include B^{ϵ}-trees [5], LSM trees [15], xDicts [6] and COLAs [1].

Recent work on write-optimized data structures [16] challenges the notion that only in-RAM data structures can keep up with high-volume data streams. Pandey et al. design WODS that can process millions of stream events per second, while also allowing efficient query performance.

Log-structured merge trees a LSM tree [15] is a write-optimized data structure that is the basis of many key-value stores. In contrast to traditional indexing data structures (such as B-trees) that implement in-place updates, an LSM tree performs out-of-place updates. It buffers updates in main memory until there are enough changes in one block of external memory to amortize the cost of the data transfer. This leads to high-throughput for updates, and improved cache performance. Popular key-value stores that are based on an LSM-tree design include LevelDB [9], BigTable at Google [7], RocksDB [11] at Facebook, and Cassandra at Apache [12]. Our system, *Diventi*, uses the popular RocksDB [11, 13] implementation of an LSM tree.

We describe how an LSM tree organizes data, performs updates (inserts and deletes) and lookups, along with its asymptotic guarantees. An LSM-tree maintains L levels, where level 0 resides in primary memory and the remaining levels are stored on disk.[1] The size of levels grows exponentially with *growth factor* T (typically T is between 10 and 20). In particular, the size of level i is T times the size of level $i-1$. The number of levels is thus $L = O(\log_T(N/M))$, where N is the total number of entries in the data structure and M is the size of Level 0 (primary, or main, memory).

A key is inserted directly in the buffer at level 0, which is implemented as a data structure called a *Memtable*. A delete is treated as an insertion of a special tombstone message. Instances of a deleted key may exist on lower levels but are not returned on queries.

When a level i fills up, runs of similar sizes on that level are merged and all runs are flushed to level $i+1$. A merge operation is also referred to as a *compaction*. When two instances of the same key merge, the newest one is kept, and the LSM tree *upserts* the previous instances, removing them from the tree. Figure 1 shows a basic LSM tree flushing two levels.

There are two main merge policies: *leveling*, and *tiering*. Leveling maintains a single sorted run at each level. Thus, whenever a run from level i is flushed to $i+1$, it is immediately merged with the run at that level. Tiering lets up to $O(T)$ (sorted) runs accumulate at each level, after which the runs are merged. Thus, leveling achieves faster lookups at the cost of slower ingestion.

On lookups, each level is searched in order until the key is found. To speed up lookups, an LSM tree may store a Bloom filter in main memory for each run. When queried, if the Bloom filter says the key is not present in the run, then the lookup operation can skip the corresponding run, improving performance.

In a tiered LSM-tree, the amortized lookup cost is $O(1 + L \cdot T \cdot \epsilon)$ I/Os, where ϵ is the *false-positive rate* of the Bloom filter. This is because there are L levels, $O(T)$

[1] Traditionally, each level in an LSM tree is implemented as a *B*-tree. However, modern systems maintain sorted runs instead and store metadata information in main-memory for each level (fence pointers that store information for every disk page of every run).

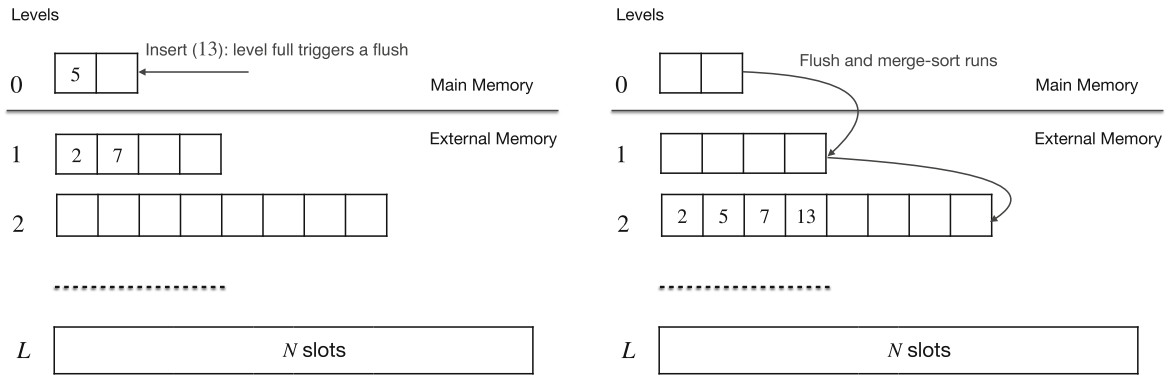

Fig. 1 Before and after a simple leveled LSM tree stores integer keys. Inserting key 13 causes the buffer to fill, triggering a flush and merge to level 1, and in turn to level 2. Parameters in this example: growth factor $T = 2$ and the size of in-memory buffer $M = 2$

runs per level, on average ϵ runs are falsely probed, and 1 run is correctly probed (assuming the key is in the LSM). The amortized update cost per element is $O(L/B)$ I/Os, where B is the block size, that is, the amount of data transferred in one I/O between main memory and external memory. Each entry is involved in at most L merge operations. When a block of B elements is involved in an I/O, each element incurs an amortized $O(1/B)$ I/Os.

In a leveled LSM-tree, the amortized lookup cost is $O(L \cdot \epsilon)$, since there are is a single run per level, and on average ϵ runs are probed. The amortized update cost per element is $O(T \cdot L/B)$. Each element, once it arrives on level $i + 1$, is involved in $O(T)$ merges, on average, from flushes from level i, each with an amortized $O(1/B)$ I/O cost. Each entry is involved in at most L levels.

The RocksDB implementation of the LSM tree uses a hybrid merge scheme, following a tiered approach for flushing from level 0 to disk, and a leveled approach for all other levels. Figure 2 comes from the RocksDB wiki [11] and shows a more detailed diagram of the actual implementation.

2.3 Related work

Low latency delete persistence in an LSM tree, deleted elements are replaced by a "tombstone" to be persisted as it naturally reaches the bottommost level through compaction and flushing. This causes unwanted space amplification and read amplification as these "deleted" elements remain on disk and may be read during queries. Sarkar et al. introduce Lethe [18], a tunable delete-aware LSM Engine, designed to persist deletes with low latency and support efficient range deletes on a secondary delete key. Lethe focuses primarily on minimizing the lifetime of these tombstones once they are created by delete operations. By contrast, Diventi's focus is on monitoring a stream of data with no explicit deletions. Events are deleted based on a global cutoff time in an I/O-efficient manner.

Elasticsearch. Elasticsearch (https://www.elastic.co) is commonly used as a storage and retrieval mechanism for generic log data, including network logs. The underlying data structures and indices, part of the Lucene library, provide generalized query support, suitable for demand queries that an analyst may want to adjust through the normal course of analysis. While this generalization is a critical asset in the analyst's toolbox, it limits Elasticsearch's ability to optimize for important cases like IP addresses. The result is such systems often requiring large clusters of servers to ingest large volumes of heterogeneous logs and handle queries.

Under the hood, Lucene creates indices at ingestion time, using datatype-specific data structures for the index. For numeric fields it uses Bkd trees; for text fields it uses an inverted index. In practice, indices are organized in what is called a "rolling" index setup, where an index name includes the date when the index was created. After a certain number of insertions into this index, a new index is created and inserted into. As space limits (or a certain number of indices) are reached, the indices with the oldest date in the name are deleted to make room for new data. This allows Elasticsearch to efficiently expire data using multiple indices.

Given that Elasticsearch is open source, freely available, and well supported by a robust community, it is commonly deployed as a security monitoring tool. Analysts frequently write analytics that use Elasticsearch. While the indexing methods are quite different from those used in *Diventi*, we believe this common use for behavioral analytics makes Elasticsearch a reasonable point of comparison to *Diventi*. For this reason, our empirical tests in Sect. 6 focus on comparing the performance of *Diventi* and Elasticsearch.

Splunk. Splunk's (https://www.splunk.com) use case is similar to that of Elasticsearch, providing a storage and retrieval mechanism for generic log data. The underlying database is proprietary, creating indexes during time of

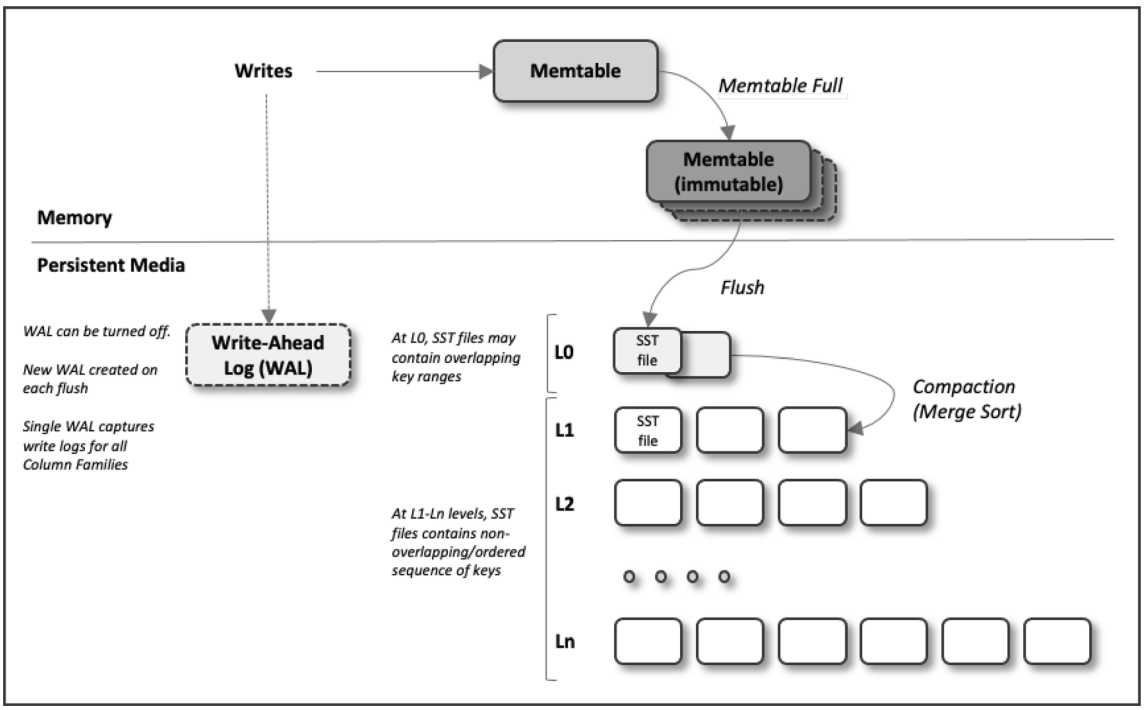

Fig. 2 Architecture of an LSM tree as implemented by RocksDB (taken from RocksDB wiki [11])

ingestion to support fast queries, storing the data in buckets.

Similar to Elasticsearch's "rolling" index setup, buckets start in the hot state, then progress through warm, cold, and frozen states as they age. Buckets in the cold state transition to the frozen state when time or size-based conditions are met. Frozen buckets are slated to be deleted.

3 Design

Here we present the design of our system, *Diventi*, to rapidly ingest and index network communication events. We first cover the way we parse connection logs and index them in our key-value store. We then discuss how we implement expiration across this key-value store to ensure that security responders are guaranteed the N most recent events, and to avoid large bursts of I/O when deleting old events.

3.1 Database design

Our primary goals in designing *Diventi* were to index our data by IP (allowing sub-second query times), while supporting high ingestion rates.

Since the majority of our data is *write-once read-maybe*, most of our computation is inserting events into our index. Our primary challenge was maintaining sufficiently high insertion rates ($> 100,000$/s) as our index grows to very large data sizes (> 50 billion events). To achieve this high insertion performance, we leveraged the strengths of WODSs, which excel at write-heavy workloads even when they grow beyond the size of RAM. We built *Diventi* around the popular key-value store RocksDB. Unlike traditional databases, RocksDB is a key-value store, so it only indexes data on a single key. Therefore we must consider how to best represent our network connection events in a key-value format. *Diventi* is designed specifically for network connection logs in a variety of sensor formats (Zeek connection logs, netflow, or IPFix [8]). These logs typically record events that involve one IP speaking with a different IP. Fields from a typical connection log are shown in Table 1. The core concept of IPs, ports, protocols and quantities of data in each direction are common across the log formats seen for security monitoring.

To turn these disparate events into key-value pairs for indexing, we must decide on a consistent key format across all log types. To allow efficient querying by IP address, we begin each key with an IP address and the timestamp. This allows us to efficiently look up all connections involving a given IP (e.g., 1.2.3.4), all connections involving an IP range/subnet (e.g., 1.2.3.X), or even to filter connections from a given IP down to a specific time range (e.g., connections with 8.8.8.8 during a given week).

One issue, however, is that simply indexing by originating IP wouldn't quite be enough, since each network event represents a connection between two IPs (originating IP and responding IP). To deal with this issue, we insert

Table 1 Some example fields from a Zeek connection log

Field	Type	Example	Description
ts	time	980997832.690939	Time of observation
uid	string	Ch80yl33lQkOymHcab	A unique hash to identify this connection
orig_h	IPAddress	209.11.146.100	Address of the originating host
orig_p	port	32113	Port number on the originating host
resp_h	IPAddress	11.254.205.104	Address of the responding host
resp_p	port	443	Port number on the responding host
proto	enum	tcp	The IP protocol field
flags	string	RSTOS0	Some flags to show the state
orig_bytes	count	84	
resp_bytes	count	20	
orig_packets	count	1	
resp_packets	count	1	

two key-value pairs for each network event, one keyed by "originating IP, timestamp" and the other by "responding IP, timestamp". This allows queries to lookup connections by either of their two IPs. To differentiate between the two copies of each key, we set an extra "reverse bit" at the end of the key if responding IP is the primary key.

A second issue is key collisions: when two identical keys are inserted, the data for one of them would be overwritten. In write-optimized data structures, insertion costs are so low that it is significantly faster to insert a key than it is to query for its existence in the tree. Therefore, to avoid data loss, instead of checking for a given key's existence we add other identifying information about the connection into our key, as shown in Table 2. Since there can only be one connection between the same two ports on the same two hosts at any given time, our keys now uniquely identify a connection and prevent issues of collisions.

With our key-format established, the remaining useful information from the connection event is formatted into a value, to create a key-value pair. The key contains IP, port, and timestamp info. The other relevant information from each network event includes connection protocols, transmission control protocol (TCP) flags, connection duration, and numbers of packets/bytes transferred in either direction, all of which get stored as the value of our key-value pairs.

Table 2 Data fields for key

Byte range	Length	Field
0–3	4	IP A
4–11	8	Timestamp
12–15	4	IP B
16–17	2	Port A
18–19	2	Port B
20–20	1	Reverse-bit/misc flags

To improve performance, we also "compress" some of the event data before storing it. Typically the original connection logs will still be available, so *Diventi* doesn't need to record all the data exhaustively. We only keep information that would be useful for a security analyst to assess if a given connection is noteworthy. Typically, analysts care more about the scale of a connection (e.g., was it 2 bytes or 2 GB?) than about precise values. To take advantage of this, we only store an 8-bit exponent for each quantity instead of its full 32- or 64-bits (i.e., we only record $\text{scale} = \log_2(orig\_bytes)$), so queried connection data is expressed in ranges such as "16–31 packets" or "1–2 GB transferred". By shrinking the data in this way, we make our key-value pairs more compact and therefore faster to index, while still giving analysts all the information they need.

The exact format of the values in our key value pairs varies based on which logs they came from (i.e., Zeek, netflow) since they all record slightly different data. The content of the *key* contains fields that are universal across a wide range of IP connection logs, so these are always consistent. However, some extra information does vary by log format, so we insert slightly different binary formats as *values*, along with an indicator of which format it's from. *Diventi* can decode the different values back to the underlying data at query time. This flexibility to ingest and index multiple log formats at the same time also makes *Diventi* a powerful tool to analyze data from many different network sensors.

While our focus has been primarily on IP data, we have clearly abstracted events (the underlying data), keys (what we want to index by), and values (event data that might be useful to an analyst). This separation has made it easy to adapt *Diventi* to several different formats of network connection data. We believe this design will also make it easy

to adapt *Diventi* to index other kinds of critical events, such as urls, e-mail addresses, or other security events.

3.2 Expiration of old data

Since *Diventi* is continuously indexing data, we want it to automatically manage its space usage, deleting old data to keep from filling the disk. Many indexing tools (Elasticsearch, for example) use multiple indices to deal with large volumes of data. Once the current index grows too large, the database "rolls over" to a new index and insertions are made in the new index. This setup ensures that the oldest data is always in the oldest index. Whenever the database grows too big, dropping the oldest index expires the oldest data.

Diventi uses a single large index to speed queries. Since our LSM tree index (RocksDB) isn't sorted by timestamp, old events are interspersed throughout the tree, mixed with recent events. To expire old data, we need to go through our entire LSM tree and delete individual events. If not implemented carefully, this operation could easily become a performance bottleneck.

To implement our expiration system efficiently, we leverage two built-in features of RocksDB: "compaction filters" and "periodic compaction."

RocksDB implements a time-to-live (TTL) feature, automatically deleting entries older than a certain age. To do this deletion efficiently, RocksDB implements a "compaction filter." As new data is continually inserted, RocksDB keeps pushing those entries into deeper levels of its LSM. A compaction is moving a set of entries down into the tree, merging them into the sorted data of the deeper level. Because data is already being read from disk, accessed, and rewritten during a compaction, there is minimal overhead to also check that data against a small function contained in the compaction filter. In this case, the function checks if the age of the data entry exceeds the TTL, and if so, drops the entry. By piggybacking on existing compaction operations in this way, compaction filters can delete old data efficiently and incur no additional I/O cost.

Enabling compaction filters alone would not guarantee that all old entries are removed: compaction filters can only delete data when that data undergoes a compaction, and compactions are intermittent. Additionally, compactions will only be performed in regions of the key-space where new data is actively being inserted. While in many workloads we expect these "opportunistic" compactions to do most of the deletion work, there are pathological workloads which leave portions of the tree un-compacted for extended periods of time, potentially allowing old entries to accumulate. To *guarantee* timely cleanup of entries, we must periodically sweep through our data store, ensuring that all entries are compacted regularly, even if they reside in a portion of the tree which is not being accessed currently. To accomplish this, we enable a second feature of RocksDB called "Periodic Compaction": which guarantees that every entry in the tree will be re-compacted at least once per a user-specified time interval. Periodic compaction can cause a significant amount of additional I/O, and in the worst case, it could require RocksDB to re-touch the entire tree. In practice, and with an appropriately-configured compaction interval, periodic compaction should have a minor impact on performance, only cleaning out the dusty corners of the LSM tree.

By using these features of RocksDB, we can delete old data opportunistically, incurring minimal I/O performance penalties despite having to delete in-place in a single index.

3.2.1 Making expiration count

We want *Diventi* to maintain its database at a configured size by deleting old entries. Using RocksDB's compaction filters as they are configured gets us partway there, but the default compaction filter only deletes old data when it reaches a specified maximum age. We wish to instead delete old data once our database reaches a desired size, i.e., once we have a total count of $> N$ entries in the tree.

Using a time-based system (i.e., time-to-live or TTL), is a common solution for discarding old data. Both RocksDB and Elasticsearch use TTL as their primary means of configuring expiration. However, using time alone has some downsides for our use case: if insertion rates are particularly low for some time, it would take less space to record the same time-span's worth of data, but a TTL-based system wouldn't take advantage of that and would leave the free space unused. If insertion rates were particularly high for a while, we'd want to start deleting data sooner since there would be more data for the same span of time, but a TTL system would not adapt. What we want is a system that expires old data while maintaining our tree *at a consistent size*.

We therefore designed our expiration system to the following constraints:

- Expiration will never delete any of the N most recent events we've indexed (for some user specified N).
- With expiration enabled, our key-value store (RocksDB) will never index more than $N + k$ events,[2] (where k is some parameter $< N$).

As we mentioned before, RocksDB provides a compaction filter which automatically deletes entries once they pass a

[2] In the analysis section, we show in detail how the value of k derives from the values of several other parameters, but in practice k should be $< N/3$.

specified age. RocksDB also allows us to override that compaction filter to change its deletion criterion. To make our count-based expiration system, we implemented a custom compaction filter that deletes entries in the tree once they are older than a global "cutoff time". We then set/adjust this cutoff time as needed to maintain our database at the desired size of N entries.

The tricky part of the problem then becomes determining what the cutoff time should be. We want a cutoff time such that the number of events in our LSM tree younger than the cutoff is as close as possible to N without going under. Determining this cutoff time is a nontrivial problem. Since our tree is not sorted by timestamp, we must construct an auxiliary data structure to help us determine an appropriate cutoff time.

3.2.2 Timestamp histogram

To specify which entries are to be expired, we need to find the timestamp of the Nth-most-recent entry, which we call the "N-cutoff" time (i.e., there are exactly N entries in the tree whose timestamp is greater than the N-cutoff).

A naive solution to determine the N-cutoff time might be to build a second tree of all entries, sorted by their timestamps, which we could then query for the Nth-most-recent. This would work, but would take up far more space and time than is needed to solve this problem.

In practice, we don't need an exact value for the N-cutoff. Since our expiration system is allowed some extra wiggle room (up to k), we can instead use a structure that provides an approximate-N-cutoff, i.e. the number of entries younger than the cutoff timestamp falls in the range $[N, N + e_{\text{hist}}]$. By allowing an approximate solution, we have a small, fast solution to the problem.

Our goal is therefore to design an auxiliary data structure with the following specification:

- Records a timestamp from each entry before it is inserted into our main key-value store.
- Can be queried for an approximate-N-cutoff, which is a timestamp such that the total number of inserted entries with timestamps newer than the cutoff is $> N$ and $< N + e_{\text{hist}}$.
- Is sufficiently accurate: (in practice: e_{hist} less than one percent of N is more than sufficient).
- Small (fits easily into a small fraction of available RAM).
- Performant for insertion: (negligible performance cost for insertions compared to the cost of inserting into the main key-value store).

We call this data structure the "timestamp histogram," since it records approximately the distribution of timestamps currently in our tree. In practice, a simple naive implementation is more than small, fast, and accurate enough.

Here is the timestamp histogram structure that we use in *Diventi* (although better designs are certainly possible). We divide time into intervals (e.g., 1 h intervals), and for each time interval we initialize one 64-bit counter. As each timestamp is inserted, we take the time interval that their timestamp falls into, and increment the corresponding counter. In effect this gives us a histogram of how many timestamps fall into each interval (e.g., each hour), over the span of time that our data covers.

To calculate the N-cutoff, we scan the counters from newest to oldest, accumulating the total number of entries we've seen so far. Once our total is $\geq N$, we can stop. We now know exactly which time interval contains the Nth most recent entry. While we don't know the exact timestamp of the Nth-most-recent, we can round up (i.e., older), and take the start of that time interval as our cutoff time. We are guaranteed to keep at least N entries alive (since we counted them). Our error, i.e., how many extra entries we keep, is at worst the maximum number of entries that fall into a single time interval. In practice, we can configure the time intervals to be small enough that this error is negligible.

We implement our timestamp histogram as an arraylist of 64-bit atomic counters, each representing a time interval. All time intervals are contiguous and the same size, so we can index into this arraylist for any given timestamp with only a bit of arithmetic. If we use atomic counters, then recording each timestamp can be easily multithreaded. When inserting a timestamp that is newer than the most recent time interval, we obtain a write-lock on the whole data structure (blocking out any other inserters), and expand the arraylist. While this is a relatively-slow operation, in practice our timestamp history is sized such that we only need to add another counter (and therefore lock the whole thing) at most once per 10s or 100s of thousands of insertions. Because each time interval only stores a single 64-bit counter, it also uses little RAM.

We briefly tested our timestamp histogram implementation on the same nodes as our main benchmarks. We tested from 2 to 64 threads, and varied interval sizes from fine to coarse. RAM usage was typically under 20 MB, and even with 1-min resolution covering 2 full years of data, (several orders of magnitude finer than needed), RAM usage only reached 60 MB. All but one test comfortably handled over 2 million events/s. Performance only dipped down to 1.5 million/s when we made intervals so fine that only 300 events landed in each, which is once again several orders of magnitude finer than needed. In our results section, we show that *Diventi* achieves sustained ingestion rates in the range of 100 to 200 thousand events/s. Because timestamp histogram can handle more than 10 times that

rate (2 million/s), we are confident that the timestamp histogram is not a bottleneck on *Diventi* as a whole.

3.2.3 Putting expiration together

We can now put together the complete expiration system. For each entry inserted into the key-value store, we also insert its timestamp into the timestamp histogram. Periodically, we query the timestamp-histogram for an approximate N-cutoff to update our global cutoff time. Our compaction filter checks against this value, ensuring all entries older than the cutoff time are deleted the next time they undergo a compaction. While the cutoff is not exact, it is guaranteed to always leave the N most recent entries. There is also an upper bound (e_{hist}) on how many extra entries it may leave un-deleted. Finally, we enable RocksDB's periodic compaction feature, which ensures that every entry is compacted at least once per a specified interval, so all old entries are removed in a timely fashion.

In the analysis section, we examine the interactions of these systems, and derive an upper-bound guarantee on the number of extra entries stored. That is, we show old entries are removed fast enough that the system never holds more than $N + k$ total entries.

4 Analysis

In this section, we prove several bounds for our expiration design. For this section, we assume the tree contains more than N entries, since expiration isn't active until we exceed N.

The expiration system allows the tree to grow beyond the target size N. We prove an $N + k$ upper bound on the size of the tree with expiration enabled, where k is a parameter derived from several parameters of our design. In practice, k is roughly the maximum number of insertions we expect to see between two periodic compaction events in RocksDB, and we can typically adjust the frequency of periodic compaction to keep k to a small fraction of N.

We also analyze the performance costs associated with our expiration system. As our expiration system features several moving parts and additions to RocksDB, we analyze the asymptotic performance of enabling expiration and show that using our expiration system should not significantly harm insertion rates over the normal LSM-tree asymptotic costs.

4.1 Analysis of excess disk usage

If expiration were instantaneous, our key-value store would never hold more than N events. However, to implement expiration efficiently, we allow our tree to grow past N. In this section, we show that the system as described in Sect. 3 never holds more than $N + k$ entries, where k is the sum of three specific tunable parameters of our expiration system.

In order to determine which entries are eligible for deletion, we store a global cutoff-time: all entries older than this cutoff can be safely deleted. If an entry has a timestamp older than the cutoff time, we call it "marked" for deletion. Entries are not physically marked, rather the global cut-off time implicitly "marks" them.

Marked entries are not immediately deleted. They are deleted during compaction. Entries undergo compaction automatically as part of normal insertion operations, and RocksDB guarantees that every entry is compacted at least once per a configurable time-period.

We can implicitly label each entry in our key-value store as follows: Entries younger than the Nth most recent are "live" and do not need to be deleted. Entries older than our cutoff timestamp are "marked" for deletion, and will be removed the next time they undergo a compaction event. Entries falling between the Nth most recent and the cutoff time are "unmarked." We are *allowed* to delete them, but the system is not yet aware of this, and so does not delete them during compactions.

By definition, there are always exactly N "live" entries and at most k marked or unmarked entries.

Theorem 1 *With expiration enabled, the Diventi system has no more than* $N + e_{\text{hist}} + I_{\text{update}} + I_{\text{compact}}$ *entries, where* e_{hist} *is the error (by construction) in setting the cutoff time,* I_{update} *is the number of entries that arrive between updates to the cutoff time, and* I_{compact} *is the maximum number of entries that arrive between and consecutive RocksDB compactions.*

Proof We first analyze how unmarked entries accumulate in our tree. Periodically, we query our timestamp histogram for a cutoff time, which becomes the new global cutoff. Ideally, this cutoff time would mark all unneeded entries for deletion, leaving only the N live entries. However, since our timestamp histogram only returns an approximate cutoff, it may leave up to e_{hist} unmarked entries. Even more unmarked events can accumulate in the time between two successive cutoff updates, as there will be up to I_{update} insertions occurring before the next opportunity to update our cutoff time.

At the moment that we update our cutoff time, the maximum number of unmarked entries in the tree is (by definition) e_{hist}. In the time between two successive cutoff updates, there will be, by definition, at most I_{update} insertions, each creating at most one additional unmarked entry. Therefore, the maximum number of unmarked entries that can ever exist in our tree is $e_{\text{hist}} + I_{\text{update}}$.

We now consider marked entries. Entries are marked if they are older than the current cutoff time, and marked entries are deleted whenever they undergo compaction. By enabling periodic compaction in RocksDB, we guarantee that each entry will be compacted at least once per a specified interval. If every entry in our tree sees a compaction once for every I_{compact} insertions into the data structure, then no marked entry can survive for longer than I_{compact} insertions before it is deleted.

Before we can put this all together, we need a third piece: consider the set of all "excess" entries in the tree, defined as the marked and unmarked entries. Changing the cutoff time may convert some entries from unmarked to marked, but will not change the set of excess entries. In addition, deletions will never add elements to the set of excess entries. The only way to create new excess entries is to insert entries into *Diventi*, If an inserted element is older than the cutoff, it is immediately in the marked or unmarked sets. Otherwise, a new entry causes the former Nth-most-recent to age out and become unmarked. Therefore, the set of excess entries can gain at most one new member for each insertion.

Now we combine these three observations to find an upper bound on k. Assume our tree contains more than N entries. Consider the state of that tree at a point I_{compact} insertions in the past, p. Each of the excess entries currently in the tree must fall into one of two categories: they were already excess entries at point p and have survived without being deleted, or they are new excess entries that have been created as a result of insertions since point p. Since we've shown that marked entries can't survive longer than I_{compact} insertions, we know that any marked entry at p has since been deleted and only unmarked entries may still be excess entries. Since we've shown that we can never have more than $e_{\text{hist}} + I_{\text{update}}$ unmarked entries in the tree at a time, at most $e_{\text{hist}} + I_{\text{update}}$ excess entries remain that were excess at point p. In addition, I_{compact} new entries have been inserted since point p, creating up to I_{compact} new excess entries.

Since each excess entry currently in the tree must either have been excess at point p or been newly created, the maximum number of excess entries is $e_{\text{hist}} + I_{\text{update}} + I_{\text{compact}}$. This allows us to bound k, the number of excess entries:

$$k \leq e_{\text{hist}} + I_{\text{update}} + I_{\text{compact}}.$$ □

4.2 The excess-storage bound in practice

In the previous section, we proved bounds on k, the number of stored entries beyond our target N, in terms of counts of insertions. For example, we define I_{compact} to be the maximum number of insertions between any consecutive compactions. However, in practice, we don't schedule these operations based on number of insertions but instead based on a fixed time interval. Scheduling in terms of time is more natural. We use several features of RocksDB, which are configured in terms of time.

We can still use the upper bound we just proved, but must now account for time and calculate k using the rates of events we expect to see.

If we know the time interval at which periodic compactions occur and the maximum rate at which new events will be inserted, then I_{compact} is the compaction interval times the maximum event rate.

Similarly, I_{update} is defined as the maximum number of insertions that can occur between two subsequent updates to the cutoff time. If we know the cutoff update time interval, then I_{update} is the update interval times the maximum event rate.

Our definition for e_{hist} is a little trickier: when querying the timestamp histogram, we receive a cutoff time. This cutoff time will always leave N events live, but it may also leave some excess events beyond N unmarked. We define e_{hist} to be the maximum number of excess events left unmarked by the timestamp histogram. The timestamp histogram divides time into evenly spaced intervals, and records how many events occurred in each interval. When querying for a cutoff time, we don't have data any more granular than these time-intervals, so we're forced to round up to the next largest time interval to ensure we keep the N most recent entries. This rounding adds excess entries younger than the cutoff time, and can include as many excess entries as there are events falling into a single time interval. e_{hist} is thus the maximum number of events that can occur within one time interval of the timestamp histogram. If we know the granularity to which the timestamp histogram is configured, as well as the maximum rate at which events can be generated, then e_{hist} is the timestamp histogram's granularity times the maximum event rate.

In summary, k is the sum of the maximum event creation/ingestion rate times each of the periodic compaction interval, cutoff-update interval, and timestamp histogram's granularity. In practice, we can set the timestamp histogram's granularity and the cutoff update interval to be quite low, such as 10 min or even 1 min. Only the periodic compaction interval contributes significantly to k, since setting this to occur more often than a few times per day decreases our insertion performance.

To put all this into practice, let's derive a disk usage bound for the *Diventi* test we describe in Sect. 5. We configured *Diventi* to expire down to $N = 25$ billion events. Periodic compaction was set to occur at 12-h

intervals, and granularity of the timestamp histogram and the cutoff update interval were both set to 10 min.

Looking at the results for this test in Sect. 6, insertion rates hovered around 175,000 events/s. At this rate, 10 min of insertions corresponds to 105 million events, so $I_{\text{update}} = 105$ million. Similarly $e_{\text{hist}} \leq 105$ million.[3] At this same rate, 12 h of insertions corresponds to 7.6 billion events, so $I_{\text{compact}} = 7.6$ billion. Adding up these three quantities, we get $k = 7.8$ billion events, so we have a guarantee that our data should never grow larger than $N + k = 32.8$ billion events. In percentage terms, this means that our *Diventi* instance with expiration set to $N = 25$ billion, should not go over 31% excess disk usage beyond what is strictly needed to store the first 25 billion events.

4.3 Asymptotic performance costs of expiration

We have shown that our expiration system should guarantee prompt reclamation of disk space. However, we still need to show that our added machinery does not hamper the performance (i.e. ingestion rates) of *Diventi*. When considering the runtime performance of *Diventi*, the biggest contributor to its performance is RocksDB, the underlying key-value store. RocksDB is an LSM tree. As described in Sect. 2, LSM trees have the following asymptotic performance characteristics:

Insert: $O(T/B * \log_T n/M)$,
Point Query: $O(\log_T n/M)$,

where B is the block size of each I/O, M is the size of the in-memory root node of the tree, T is the size-ratio between levels of the LSM tree, and n is the total problem size.

Since *Diventi* delegates most of the data-structure work to RocksDB, we need only ensure that nothing *Diventi* does will affect the asymptotic performance of RocksDB. When inserting entries, *Diventi* merely parses them from the input logs and passes them off to RocksDB for insertion, so *Diventi* adds $O(1)$ time per insertion and doesn't affect the asymptotic insertion performance. Similarly, when executing a query, we rely on RocksDB to do all the heavy lifting and merely reformat the data we receive, again a constant-time operation.

For the expiration system, we consider the costs of adding a compaction filter, of enabling periodic compactions, and of inserting into and querying the timestamp histogram.

- *Compaction filter* our compaction filter runs for each entry in the tree, each time it is compacted (i.e. moves down a level). However, this compaction filter only runs when entries are already in cache, and the check itself is constant-time (checking a timestamp against a global), so it does not increase the asymptotic cost of compaction.
- *Periodic compaction* in the worst case, this forces us to re-write the entire tree periodically, costing N/B I/Os each time. Therefore, for every N insertions, our periodic compaction must only occur $O(\log_T n/M)$ times. This makes the total compaction cost comparable to the insert cost for the N insertions: $O(N/B * \log_T n/M)$. This is the only cost that constrains us in any significant way in practice.
- *Timestamp histogram* the timestamp histogram for expiration is a black box, permitting a variety of implementations. For an asymptotically efficient implementation of the timestamp histogram, we use a second LSM-tree, indexed by event timestamps. Keeping a second LSM-tree doubles the insertion cost, which leaves the asymptotic cost unchanged. We can query this LSM-tree for a cutoff time using binary search across the $O(\log_T n/M)$ sorted runs of data, taking $O(\log^2(n))$ per query. We would only need to run this query a constant number of times per N insertions, so querying an LSM-based timestamp history would not pose a significant performance cost.

  In practice, an LSM-tree-based implementation is far larger and slower than we need. In our experiments, we used a small, simple array list of counters. At the data sizes we used, our implementation fit in under 20 MB of RAM, and ran fast enough on both inserts and queries that it had no noticeable impact on *Diventi* 's performance.

With appropriate choices for *Diventi* parameters, the final asymptotic performance cost per insertion remains equal to that of the underlying LSM tree.

5 Methodology

While we have put *Diventi* through its paces in real-world deployments, we wanted to run controlled benchmarks to gauge its performance in a consistent environment and to compare it head-to-head with an existing indexing tool, Elasticsearch. We used a set of anonymized real-world network traces collected during the 2019 SuperComputing Conference. These traces provide 2.4 billion connections,

[3] e_{hist} This is slightly oversimplifying: actually I_{compact} and I_{update} both depend on ingestion rate, whereas e_{hist} actually depends on the rate of event creations, as determined by their timestamps. In a practical system, these two rates will be the same, as events should be ingested at the same rate as they are created. However, in the case of our test setup, we are ingesting a pre-generated backlog of data as fast as possible, so the ingestion rate is significantly higher than the event-creation rate.

but by replaying them in a loop we created a synthetic workload representing 100 billion connections.

We ran our tests with this synthetic workload on a cluster of identical Linux machines. We recorded insertion rates and disk usage over the course of these tests. We also ran several sets of queries against these test instances to gauge query performance.

5.1 Real world traces

During SC 2019 as part of the NRE experimentation track, we collected and anonymized traces of network communications, specifically connection logs, along with attributes of the original IP including its country code, whether access was over WiFi, and if it was blocked and why. These logs consisted of more than 2.4 billion connections covering approximately 4.2 million unique IP addresses. To preserve anonymity, each IP address is remapped to a different IP. This remapping is consistent throughout the dataset and randomly generated, with the restriction that internal IPs were all kept under 255.255.* and 255.254.* and external IPs were mapped to everything else. These events span traffic from 10/30/2019 through noon MT on 11/21/2019. This dataset has been made publicly available as a part of this paper's publication.

5.2 Extending traces for a large-scale test

When working with large-scale data structures, performance often strongly depends on how large the data grows. To collect consistent measurements we need a repeatable workload that represents months or more of data. While our traces from SC 2019 were relatively large, they represented four weeks of activity, with two weeks low usage, one week moderate usage and one week heavy usage. By looping these traces approximately 40 times, to 100 billion events, with timestamps adjusted appropriately, we created a synthetic workload of roughly 1.8 TB of gzipped Zeek logs. It would be fair to extrapolate that 100 billion events represents something greater than 40 weeks of activity for a typical enterprise. We considered several other options for synthetic traces and did generate a set of traces with uniformly random IP addresses. In the end any synthetic traces would have limitations (loss of temporal locality, uniqueness/distribution of IPs, etc.). We believe the looped SC dataset gives a good balance between realism and simplicity.

Our synthetic workloads have two limitations. First, after the first loop, no new IP addresses arrive. Thus, the temporal patterns do not accurately represent months of data. Even though there are no new IP addresses, each event has a new timestamp, forming a new (key,value) pair, which we store.

The second, potentially major, concern with simply looping the data is that it might be cached after the first loop, causing an unintended boost to performance, or that it might cause the system to simply retread the same access patterns each time it repeats. However we believe this isn't a major factor: a single instance of the SC 2019 trace takes hours to ingest and is over 160 GB uncompressed, while our test machines were limited to 128 GB of RAM, only 64 GB of which was allocated to *Diventi* and Elasticsearch. Furthermore, the addresses accessed are spread throughout the tree which, by the end of the test, will be terabytes in size. While the access pattern is the same on each loop with respect to IP address, the pattern of disk accesses should be very different on each iteration as the tree grows, shifting and spreading further apart.

Thus, we believe these limitations are relatively minor, and we believe our looping dataset captures the most important aspects of a real-world workload for testing database performance: spatial and temporal locality of incoming IP addresses, distribution from which IPs are drawn, and burstiness of incoming events on short timescales.

Nevertheless, to provide a broader perspective, we also present results using a uniformly random IP workload of 50 billion connection events. While random workloads are also not representative of real workloads, they do not suffer from excessive data locality, and stand as a point of comparison against our looping dataset. We generated 50 billion Zeek events with IPs drawn uniformly randomly from the IPv4 space. Due to the less realistic nature of the random dataset, we ran a smaller suite of tests.

5.3 Our test environment

We ran our tests with these workloads on a cluster of identical machines, configured with 16 cores and 128 GB of RAM, with NVME SSD storage. The physical machines we used were Dual Socket AMD Epyc 7601 2.20 GHz CPUs (64 total cores) and 1 TB of RAM, but to achieve a more "modest" hardware setup we used Linux *cgroups* to limit the RAM and core count available to *Diventi* and Elasticsearch. The storage used for these tests was a large pool of NVMe SSDs in RAID 0, organized as an *XFS* filesystem. All machines were running CentOS 7 Linux.

We used the command *cgset* to limit memory to 128 GB of RAM and restrict CPU use to the first 16 cores. We then used *cgexec* to run our instances of *Diventi* and Elasticsearch under those cgroup restrictions. Note that setting the memory limit in cgroups restricts the total memory used across all processes running in that cgroup, as well as across all their child processes. The memory limit also includes "free" memory used by the kernel as filesystem cache for those processes.

We configured our *Diventi* tests as follows: expiration was either disabled or set to keep 25 billion events ($N = 2.5 \times 10^{10}$). Periodic compaction was configured to run every 12 h. Our timestamp histogram was configured to 10-min granularity, and expiration cutoff was set to update at 10-min intervals. *Diventi* was configured with 20 threads to insert into RocksDB. RocksDB was tuned as follows: it was allocated 64 GB of RAM for dedicated caching (half of our total 128 GB). It was allowed an extra 16 GB of RAM for memtables. It was configured with 12 threads for compactions and flushes. We increased the RocksDB *pending_compaction_bytes* limit to a 128 GB soft limit and 512 GB hard limit.

We performed a basic configuration of Elasticsearch as follows: we ran a single instance (one shard) using 32 GB of RAM, as this was the maximum amount recommended in the documentation. We created an Elasticsearch index with an explicit mapping corresponding to the fields of the Zeek logs we used in our benchmarks. We disabled indexing on all but the source IP and response IP fields. We used Logstash to parse our Zeek log-files and insert them into Elasticsearch. We configured Elasticsearch's Index Lifecycle Management to rollover indices after they reached 50 GB in size, as recommended in the documentation.

To configure expiration at 25 billion events in Elasticsearch, we had to approximate: Elasticsearch does not appear to have a feature for deleting old indices once the total number of inserted events exceeds a certain count. To approximate this, we set indices to be deleted after they were 8 days old, which, given Elasticsearch's very consistent insertion rates over the course of the test, kept the database size in the range of 23–26 billion events.

5.4 The tests

For both *Diventi* and Elasticsearch, we ran the following tests:

- *Diventi*, ingesting 100 billion events from *SC Looped*, no expiration,
- *Diventi*, ingesting 100 billion events from *SC Looped*, expiring down to 25 billion events,
- Elasticsearch, ingesting 100 billion events from *SC Looped*, no expiration,
- Elasticsearch, ingesting 100 billion events from *SC Looped*, expiring after 8 days (23–26 billion events),
- *Diventi*, ingesting 50 billion events with random IPs, expiring down to 25 billion events,
- Elasticsearch, ingesting 50 billion events with random IPs, expiring after 8 days (23–26 billion events).

To measure ingestion performance, we recorded counts of total events ingested so far by both *Diventi* and Elasticsearch, at 10-s intervals. We also recorded the total disk usage of both *Diventi* and Elasticsearch at 10-s intervals.

To measure query performance, we ran several sets of test queries against *Diventi* and Elasticsearch. We put together two sets of 1100 IP addresses: 1000 distinct IPs chosen at random from our test workload, as well as 100 random IPs not in our test dataset (to test queries with an empty result). We performed these queries using a python script to query for those IPs, communicating with *Diventi* and Elasticsearch over an HTTP socket on localhost. For each IP, we recorded the total time taken for each query from first HTTP request until all data was received by python, including the time taken to parse and requery for subsequent pages of results (in the case of large, paginated queries).

We performed these query tests twice on each test instance. We first ran our queries partway through the test runtime, while events were actively being ingested, in order to simulate query behavior in an active instance of *Diventi* or Elasticsearch. We also re-ran the queries at the end of the test, once the databases had reached their maximum size and were idle. To prevent issues arising from caching, each set test of IPs was only queried once. We switched to the second set when repeating queries on the same test instance.

For the mid-run queries the primary goal was to look at the effect of a running vs. idle database; the precise size of the database was less critical, so our queries were run at slightly different instances in time: for Elasticsearch we performed mid-run queries after around 43 billion events inserted; for *Diventi* they fell closer to the 75 billion-event mark.

On the random-workload tests, almost all IPs appeared very few times, therefore query tests on the random workload added several explicit range queries to ensure we could test larger query responses.

6 Results

We present the results from our empirical tests. Our query results show that for common queries (those with fewer than 1000 results) Elasticsearch typically responded in 7.7 s, while *Diventi* averaged less than 46 ms, more than 2 orders of magnitude faster. It is responses in the range of milliseconds that provide a foundation for behavioral analytic systems that can not only alert but also respond to threats.

Our *Diventi* servers ingested and indexed logs more than 4 times faster than Elasticsearch. *Diventi* ingested 100 billion events in 6.2 days, settling to a stable ingestion rate of over 171,000 events/s. The Elasticsearch tests took over

32 days to ingest the same 100 billion events, at a consistent ingestion rate of 37,000 events/s. Our system with automatic expiration had no noticeable overhead and maintained a consistent ingestion rate between 171,000 and 183,000 events/s. The system without expiration continued to slow to 116,000 events/s as a result of the increasing size of the database. Additionally, we show empirically that, with expiration, our disk usage remains stable while the system continues to ingest data. We present this data as a series of graphs of individual runs followed by tables to compare and summarize the data.

6.1 Ingestion performance

6.1.1 Ingesting 100 billion events without expiration

Figure 3a shows *Diventi* 's insertion rates over the course of ingesting our 100-billion event workload (without expiration). The X-axis shows total insertions. The Y-axis shows the insertion rate, in thousands of events per second. The blue points show average rates over 10-s intervals, the red segments show rates averaged over 1-h intervals, and the black lines show averages over a full sixth of the test's runtime. The blue points show a strongly bimodal behavior in insertion rate, alternating between inserting rapidly (>400 k/s) and inserting very slowly (<100 k/s). Looking at the longer-term averages, we can see a clearer trend over the course of the test, starting with sustained rates of >400 k/s for the first few billion events, dropping to 170 k/s at 25 billion events, and drifting down to 116 k/s by the time we have ingested the full 100 billion events.

We believe the bimodal behavior is due to RocksDB's write-stalls, which are how it throttles insertion rates when I/O-bound. When insertions arrive faster than they can be written to disk, RocksDB triggers a write-stall until the backlog clears. This means that ingestion happens in bursts, but the average ingestion rate over time should follow I/O performance, which is reflected in Fig. 3a.

Figure 4a shows Elasticsearch's insertion rates for the same 100 billion event workload (without expiration). The X-axis shows total insertions. The Y-axis shows the insertion rate. Figure 4a shows that Elasticsearch was inserting at a steady 34 k/s over most of the test's runtime. The brief performance blip near the 40 billion insertion mark coincides with when we were querying Elasticsearch, so we believe that is due to the additional overhead of answering queries while ingesting data.

Elasticsearch's insertion rates did not decrease over time as *Diventi* 's did. We believe this is due to Elasticsearch keeping many small indices, and rolling over to a new one whenever its current index gets too big. By keeping its indices small, Elasticsearch maintains its insertion rate as the database grows, but as we show in Sect. 6.2, this results in a significantly worse result for query response times.

6.1.2 Ingesting 25 billion events with expiration

Figures 3b and 4b show ingestion rates for *Diventi* and Elasticsearch, respectively, with expiration enabled. Expiration was configured to keep the 25 billion most recent events indexed. The dashed line shows the 25-billion-event point when expiration began.

Neither Fig. 3b nor Fig. 4b show any significant performance degradation once expiration is enabled. This is particularly important for *Diventi*, since *Diventi* 's expiration system can't simply drop the oldest index, but must actively delete old events interspersed through its tree. The lack of performance drop confirms that *Diventi* 's expiration system works efficiently. Also, Fig. 3b shows *Diventi* 's insertion rate no longer decreases as the test goes on. Instead of sinking to 116 k/s, *Diventi* keeps inserting at >170 k/s for the entire test, since the expiration system keeps our data size constant at a little over 25 billion events.

For Elasticsearch, Fig. 4b shows a consistent performance of 32–37 k/s (note: in this figure, Elasticsearch also exhibits the bimodal behavior, alternating between approximately 70 and 0 k/s, but since we collect data every 10 s and Elasticsearch updates at a 15 s interval we believe this is merely an anomaly in our data collection).

6.1.3 Summary and expiration

The key-takeaway from our ingestion-rate data is the following: by using a single write-optimized index, and focusing on indexing the core network communication keys critical to security monitoring (IP addresses and time), *Diventi* is able to maintain insertion rates $4\times$ faster than Elasticsearch. While Elasticsearch continues to perform the important task of enabling generalized queries on network flow data, we believe a precision tool like *Diventi* creates a stronger platform for building advanced behavioral analytics that must operate in near real-time for large data sets.

Moreover, we show that the overhead from expiration in *Diventi* is not noticeable and far outweighs the impact of a larger database. As *Diventi* expires on a single index, performance depends on expiration overhead and database size. *Diventi* has a negligible expiration overhead and performance is stable while expiring. By restricting the size of the database, we prevent performance loss.

6.1.4 Ingesting random IP workload

Our second workload uses uniformly random IP addresses as described in Sect. 5.2. While random data is less

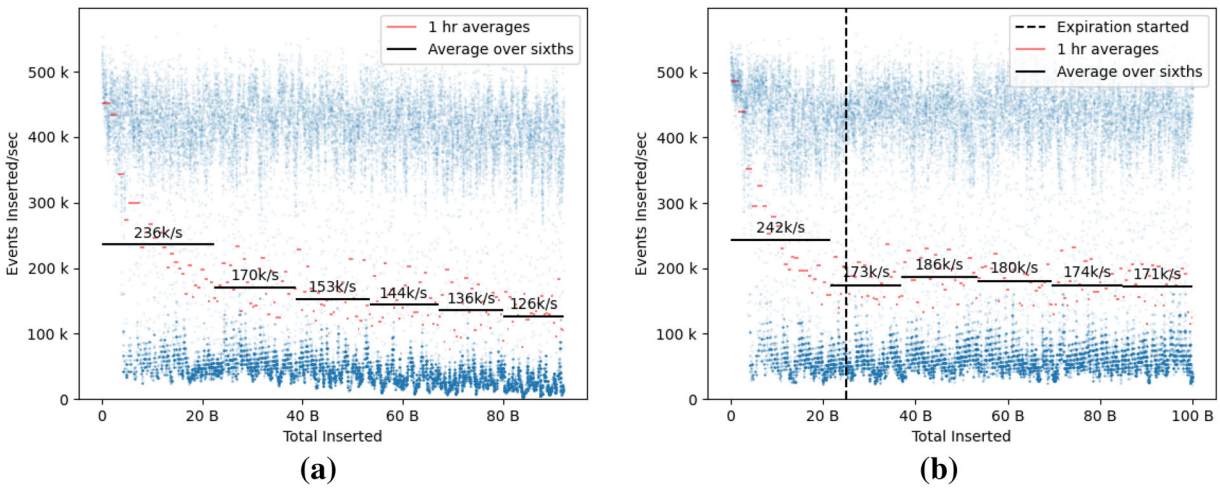

Fig. 3 Insertion rates for *Diventi*, ingesting 100 billion events from our *SC Looped* workload. **a** No expiration. The bimodal performance on short timescales is due to RocksDB's write-stall behavior when I/O-bound. **b** Expiring to 25 billion events

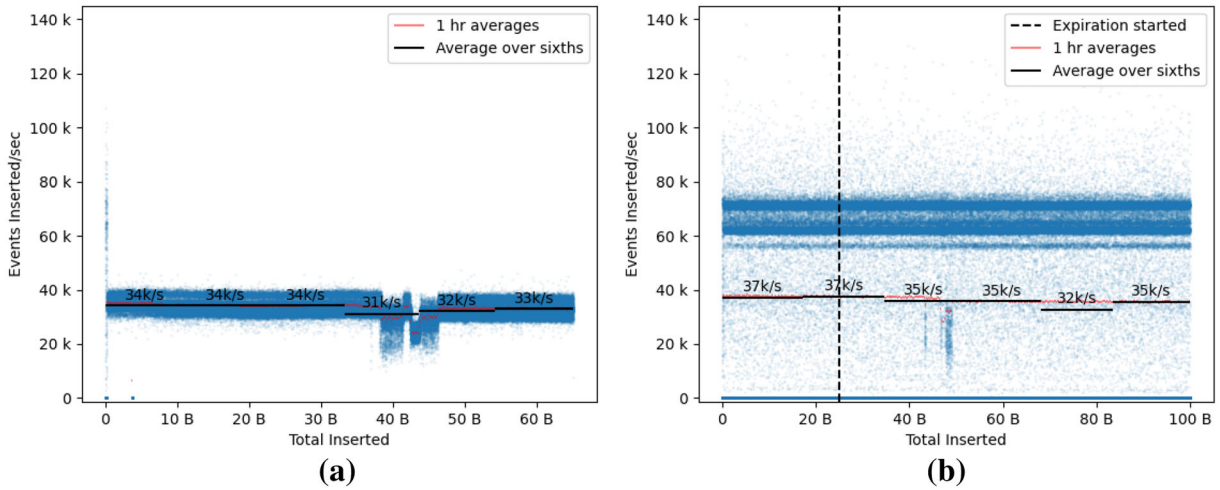

Fig. 4 Insertion rates for Elasticsearch, ingesting 65 billion events from our *SC Looped* workload. **a** No expiration. The performance anomaly near 40B coincides with when we were querying Elasticsearch. *Note* test was cut short at 65 billion events due to running out of disk space. **b** Expiring to 25 billion, ingesting 100 billion events total

reflective of real workloads, it approximates worst-case behavior in terms of data locality.

Figures 5 and 6 show ingestion rates on our random-IP dataset for *Diventi* and Elasticsearch, respectively. These also had expiration enabled to 25 billion events. The graphs show performance similar to runs on the *SC Looped* dataset. *Diventi* stabilizes at an ingestion rate of 145 k events/s, roughly 20% slower than its performance on *SC Looped*. Elasticsearch maintains an ingestion rate of 35 k/s, right in line with its ingestion rates on the other tests.

Comparing Fig. 5 to Fig. 3b, we observe some additional qualitative differences between *Diventi* running on random data and running on looped real-world data. The 1-h averages (the red segments) show that the random workload maintains much more consistent ingestion rates over time. With the looped dataset (Fig. 3b) in steady-state, the 1-h averages vary from 140 k to 230 k. With the random dataset (Fig. 5), they stay within the range of 130 k to 170 k. We speculate that *SC Looped* varies more wildly due to more locality in the dataset: when a burst of events arrives from the same IPs or from IPs close together in the tree, ingestion rates should speed up due to better cache performance. However, we would expect the random dataset to have little data locality, and so would run slowly and steadily for the whole test.

The blue points (10-s averages) show a pattern of horizontal lines. As mentioned earlier, we believe these lines are due to patterns of insertions and write-stalls in RocksDB. Since the random workload is uniform over time, these bands show up clearly.

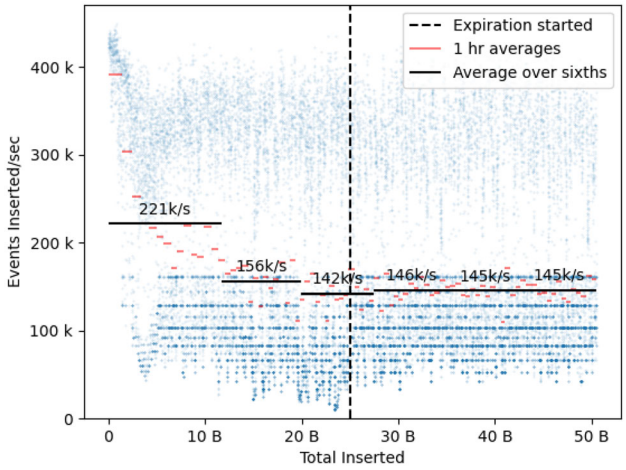

Fig. 5 Insertion rates for *Diventi* running on random-IP dataset, expiring to 25 billion

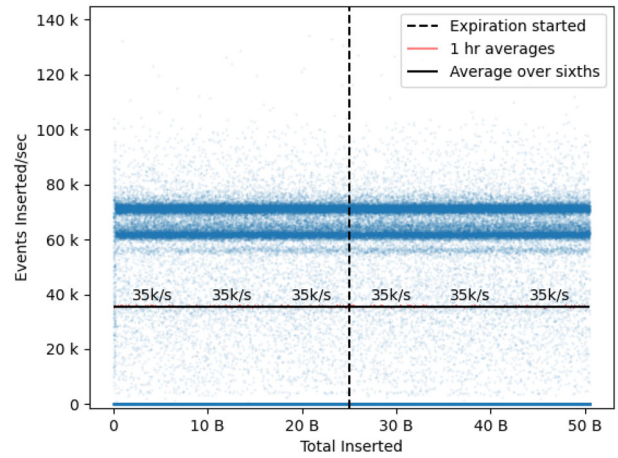

Fig. 6 Insertion rates for Elasticsearch running on random-IP dataset, expiring to 25 billion

6.2 Query time

The original purpose of designing *Diventi* was to be able to query network logs in milliseconds. To test query performance, we prepared a set of 1100 IPs to query against our databases (as described in Sect. 5.4), and recorded the time to complete those queries at the command line.

6.2.1 Querying at 25 billion events with expiration

Figure 7 shows time taken to query our two databases for each of a set of 1100 test IP addresses. Queries were run as *Diventi* and Elasticsearch continued to ingest data, which is how we expect would expect one to use *Diventi* in a real-world deployment. Expiration was enabled, actively keeping both databases to 25 billion events indexed.

Each point represents a single query. The X-axis shows the size of the query result, i.e., number of events returned

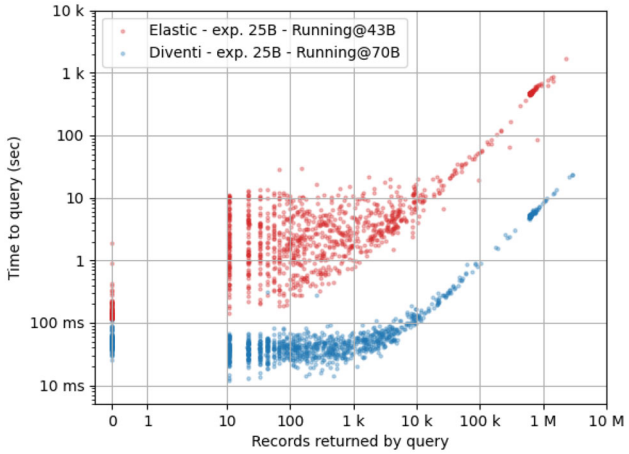

Fig. 7 Query times vs. number of records returned. Test run with *Diventi* and Elasticsearch actively ingesting data, with expiration set to keep 25 billion events. This test was run when Elasticsearch and *Diventi* had gone through a total of 43 billion and 70 billion events, respectively

by it. The Y-axis shows the time taken for the query to complete. Both scales are logarithmic. Figure 7 shows that for large queries (>10 k records), query times are proportional to result size for both *Diventi* and Elasticsearch. Elasticsearch takes over 11.5 min (704 s) to respond to a query returning a million events, whereas *Diventi* responds in 8 s, over 80$\times$ faster. On smaller queries, (<1 k records), query time is not significantly impacted by the result size. Elasticsearch completes these small queries in 3.1 s on average, whereas *Diventi* completes them in 40 ms. Elasticsearch answers 0-size queries (no match) in 180 ms on average while *Diventi* takes 50 ms on average. We suspect this performance improvement is due to Elasticsearch using some sort of Bloom filter to answer negative queries rapidly.

6.2.2 Querying at capacity without expiration

Figure 8 shows query times for *Diventi* and Elasticsearch filled to capacity. In this test, expiration was disabled and databases were idle, having filled up to 65 billion events for Elasticsearch and 100 billion for *Diventi*. The X-axis shows the size of the query, i.e., number of events it returns. The Y-axis shows the time taken for the query to complete. At idle, Elasticsearch's performance on large queries is significantly improved, taking only 6.5 min to return a million events. However, its performance on small queries is about 3$\times$ worse, taking a full 9.1 s on average to respond to queries under 1 k records. Additionally, Elasticsearch's 0-result performance is back on par with that of a typical query, taking 7 s to complete on average.

We suspect the improvement in large-result performance comes from Elasticsearch being idle for this test,

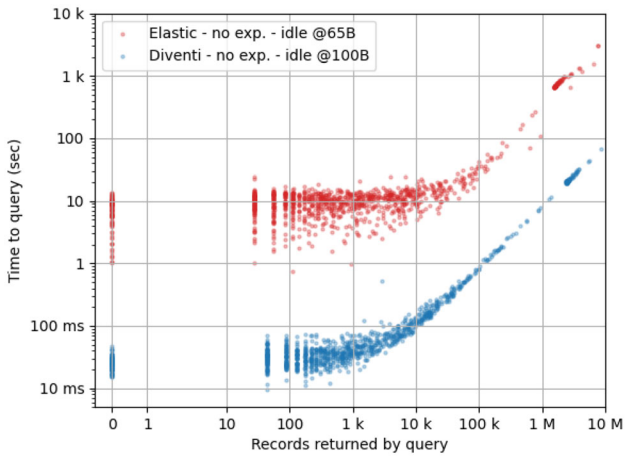

Fig. 8 Query times vs. number of records returned. Test run with Elasticsearch and *Diventi* both at idle, indexing 65 billion and 100 billion events, respectively

since large results are primarily I/O-bound, and depend little on the current size of the tree. However, as Elasticsearch uses multiple indices, it must perform multiple queries instead of a single query, causing a potential slowdown. Comparing with results from several other query tests (see Tables 3, 4), it appears that the performance hit on small queries is primarily due to the increased size of the database, with Elasticsearch holding 65 billion events in Fig. 8, up from 25 billion in Fig. 7. For the slowdown of 0-size queries, we suspect that this may be due to Elasticsearch using something like Bloom filters to accelerate queries, and those filters losing their effectiveness once data grows beyond a certain size.

6.2.3 Querying random IP workload

We ran limited query tests against our random workload to see if query performance differed significantly between looped and random data. Figure 9 shows both query times for *Diventi* and Elasticsearch after ingesting 50 billion events and expiring at 25 billion events. Since all IPs in the random data appeared with similar frequency, we did not get a continuum of query response sizes in our results. Therefore, to achieve a more diverse set of data points, we performed queries on random IP address ranges, with subnets ranging in size from /31 to /16 in CIDR notation. This establishes the query performance for larger results, but leaves Tables 4 and 3 with gaps for query-sizes that we did not test.

In *Diventi* 's case, queries with larger results ($>$ 100 k records) were only 10% slower on random than on looped, but smaller results saw a 70% slowdown on random data. For Elasticsearch, running on random data slowed down larger results only 15% over looped, but smaller results were more than 3 times slower.

6.2.4 Comparison, summary, and expiration

To further examine our query results we created two tables. The first, Table 3, shows total query response times for queries with less than 10,000 results. For queries with more results we find it useful to divide the total query response time by the number of records returned. Table 4 shows this metric as seconds taken per million results. For the 1000–10,000 results column, the startup costs still dominate. For the larger-result columns, we see two key trends. *Diventi* responds rather consistently independent of whether it's ingesting or idle. For Elasticsearch, the additional

Table 3 Average query times in milliseconds, for small queries

	Test			0-Size queries (ms)		1–1 k Queries (ms)		1 k–10 k Queries (ms)	
a	*Diventi*	25B	Idle	23.3	(±7.4)	21.1	(±6.6)	47.0	(±19.1)
b	*Diventi*	100B	Idle	25.6	(±6.4)	33.8	(±11.5)	66.9	(±39.3)
c	Elastic	25B	Idle	131.7	(±36.2)	1040.9	(±995.4)	1804.7	(±1073.2)
d	Elastic	65B	Idle	7157.2	(±3265.3)	9176.2	(±3455.2)	10,260.1	(±3402.5)
e	*Diventi*	25B	Running	51.2	(±14.8)	40.0	(±18.9)	66.7	(±28.2)
f	*Diventi*	70B	Running	40.6	(±8.3)	45.6	(±14.3)	72.3	(±27.2)
g	Elastic	25B	Running	183.6	(±195.8)	3127.4	(±3308.9)	5429.1	(±3711.7)
h	Elastic	43B	Running	2907.2	(±2743.4)	7736.4	(±5580.8)	9772.0	(±5575.1)
i	*Diventi* (rand)	25B	Idle	N/A		28.2	(±8.3)	49.0	(±15.9)
j	Elastic (rand)	25B	Idle	N/A		3949.2	(±1290.0)	5549.5	(±1369.1)

Standard deviation in parentheses. Results in 1 k–10 k column have high standard deviations, since query time starts becoming linear in the number of records

Table 4 Average query times in seconds per million records, for large queries

	Test			1 k–10 k (s/M rec.)		10 k–100 k (s/M rec.)		> 100 k (s/M rec.)	
a	*Diventi*	25B	Idle	16.0	(±5.5)	8.8	(±1.1)	7.6	(±0.2)
b	*Diventi*	100B	Idle	23.0	(±15.3)	10.0	(±1.8)	8.3	(±0.4)
c	Elastic	25B	Idle	688.9	(±509.6)	281.8	(±83.3)	381.1	(±69.0)
d	Elastic	65B	Idle	4032.6	(±2851.6)	674.3	(±371.6)	385.2	(±73.7)
e	*Diventi*	25B	Running	23.8	(±10.7)	10.4	(±1.4)	8.3	(±0.4)
f	*Diventi*	70B	Running	24.7	(±12.2)	10.4	(±1.9)	8.1	(±0.7)
g	Elastic	25B	Running	1855.9	(±1741.7)	687.6	(±256.0)	704.0	(±111.8)
h	Elastic	43B	Running	3639.3	(±3414.7)	785.9	(±348.3)	693.3	(±131.9)
i	*Diventi* (rand)	25B	Idle	16.5	(±5.6)	9.3	(±1.0)	8.1	(±0.2)
j	Elastic (rand)	25B	Idle	2077.4	(±1025.5)	533.3	(±151.1)	398.1	(±29.1)

Standard deviation in parentheses. Results in the 1 k–10 k column have very high standard deviations, since query times start being non-linear with number of records

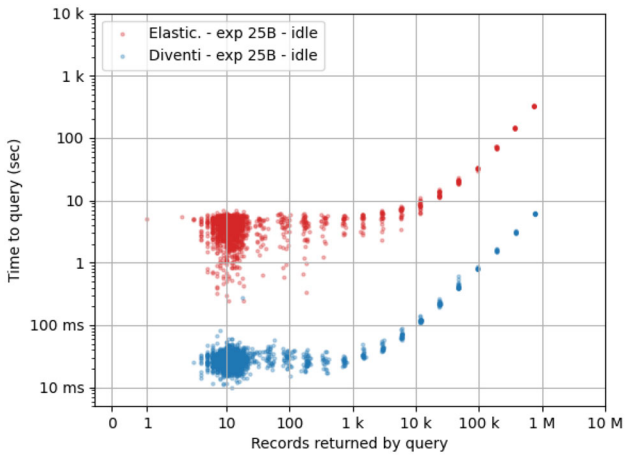

Fig. 9 Query times vs. number of records returned. Test run on random-IP workload, with Elasticsearch and *Diventi* both at idle, indexing 50 billion and expiring at 25 billion

I/O of ingesting causes a significant loss in query response time.

Overall, query response times for *Diventi* are as much as 100 times faster than Elasticsearch. For modestly sized queries (<1 k records) *Diventi* responds to most queries in under 80 ms (mean $+ 2 \times$ standard deviation). For large queries (>1 million records), *Diventi* responds in under 10 s. Additionally, *Diventi* 's query times stay low even as our database size grows, increasing only 30% as our data quadrupled in size. We believe this performance benefit is due to *Diventi* storing data in a single comprehensive index, so once the point is found all results are sequential reads. For small queries, however, query time is dominated by the time to traverse the index. We believe our improvement on small queries is due to *Diventi* only having to search a single tree, while Elasticsearch has to traverse potentially many trees. By using only a single on-disk tree, *Diventi* 's query times scale logarithmically with no hard limit in sight.

Expiration speeds up querying in *Diventi* by keeping the database small. As expiration does not introduce additional indices but maintains a single one, expiration does not positively or negatively effect query time outside of its effect on disk usage.

6.3 Disk space usage

A major feature of *Diventi* is its ability to efficiently expire data out of its single index. We recorded the total disk usage of *Diventi* over the course of the tests to better understand how our expiration algorithm behaved in practice.

Figure 10 show's *Diventi* 's disk usage with and without expiration enabled. The X-axis shows total insertions. The

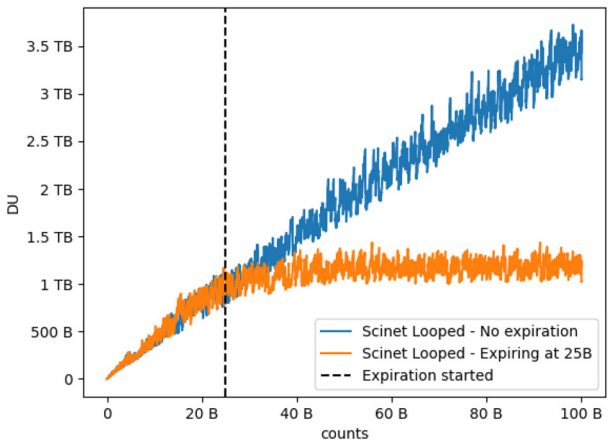

Fig. 10 *Diventi* 's disk usage, with and without expiration enabled. From 24 to 26 billion insertions, both graphs fluctuate in the range of 0.76–1.17 TB in size. Without expiration, disk usage rises as high as 3.73 TB by the end of the test. With expiration enabled, the disk usage never exceeded 1.44 TB staying effectively less than 25% excess

Y-axis shows total disk usage. The dashed line shows when expiration is enabled at 25 billion events. Without expiration enabled, disk usage increases in proportion to total events inserted. Disk usage oscillates within a range. We believe this oscillation is due to the LSM-tree compaction mechanisms used by our underlying database, RocksDB, as well as the effects of compression changing as data is shuffled around. Between 24 and 26 billion insertions, disk usage for both tests oscillated in the range of 0.76–1.17 TB. Without expiration, the maximum disk usage by the end of the test was 3.73 TB.

With expiration enabled, however, Fig. 10 shows that disk usage levels off after expiration is enabled, increasing only slightly past its value at the 25 billion mark, and staying stable throughout the rest of the test. For the entire second half of the test from 50 to 100 billion insertions, disk usage varied between 1.0 and 1.44 TB. This is consistent with our analysis in Sect. 4.2, and shows that our expiration system is working as intended.

7 Conclusion

We present an approach to organizing cybersecurity monitoring data using write-optimized data structures that leverage non-volatile storage to support large datasets, optimize data ingestion and query performance, and efficiently expire older data to enable sustained operations within a fixed disk space. This approach was implemented in a tool called *Diventi*. We extended real-world traces to create a 100 billion event workload in order to benchmark *Diventi* and Elasticsearch in similar environments. Beyond our benchmark, *Diventi* has seen live operational use in multiple environments. *Diventi* supported analysts' monitoring efforts and provided a foundation from which analysts were able to build more advanced rapid response behavioral analytics.

Using our benchmark dataset, we compared *Diventi* with a reasonably configured Elasticsearch instance. *Diventi* typically ingested events more than 4 times faster than Elasticsearch and delivered query responses on the order of 100 times faster. This makes *Diventi* well suited for frequently executed security monitoring queries where query response time should be minimized, such as for automated response and data enrichment, while still leaving the critical role of enabling generalized queries to tools like Elasticsearch. Our analysis and benchmarks showed that the overhead from our expiration had no noticeable negative impact on ingestion or query performance. By introducing an efficient expiration approach that automatically deletes old data, *Diventi* can run indefinitely in an operational state without halting the processing of new data or requiring the addition of more drive space.

The concepts and approaches described were oriented toward deploying a single instance of *Diventi* in an operational environment for cybersecurity monitoring of network connection events. Such focused systems bridge the gap between sensors and analytics. They provide a foundational building block that enable security responders to easily build behavioral analytics that consider months of data in seconds and respond to threats as they occur. This approach is valid for nearly any security monitoring domain involving large, mostly homogeneous datasets with readily indexable primary keys. Furthermore, while the standalone nature of *Diventi* makes it easy to deploy, the approach could also scale out, for example deploying multiple *Diventi* instances in a distributed architecture across multiple sensors.

Acknowledgements Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under Contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Author Contributions All authors contributed to the concepts, design, analysis and experimentation planning. JV and DRD lead the implementation and testing and were supervised by EDT and TMK. The initial conception and inspiration for this work was a team effort as was the writing and revisions.

Funding This research was funded by Sandia National Labs LDRD 218336 and 222383. We also gratefully acknowledge support from NSF Grants CCF-2118832, CCF-2106827, CCF-1725543, CSR-1763680, CCF-1716252, CNS-1938709.

Data availability The anonymized traces from SuperComputing and source code for our IP address indexing tool and our test result data will be made publicly available with this publication.

Code availability Our code and associated tools will be made publicly available with this publication.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

References

1. Bender, M.A., Farach-Colton, M., Fineman, J.T., Fogel, Y.R., Kuszmaul, B.C., Nelson, J.: Cache-oblivious streaming B-trees. In: Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 81–92. ACM (2007)
2. Bender, M.A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B.C., Porter, D.E., Yuan, J., Zhan, Y.: An introduction to B^{ϵ}-trees and write-optimization. Login USENIX Mag. **40**(5), 22–28 (2015)

3. Berry, J.W., Porter, A.M.: Stateful streaming in distributed memory supercomputers. In: Slides from an Invited Talk at the Chesapeake Large-Scale Analytics Conference (CLSAC), October 2016. Slides archived at OSTI (2016). https://www.osti.gov/servlets/purl/1406959. Accessed 3 April 2018
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
5. Brodal, G.S., Fagerberg, R.: Lower bounds for external memory dictionaries. In: Proceedings of the Fourteenth Annual ACM–SIAM Symposium on Discrete Algorithms, pp. 546–554. Society for Industrial and Applied Mathematics (2003)
6. Brodal, G.S., Demaine, E.D., Fineman, J.T., Iacono, J., Langerman, S., Munro, J.I.: Cache-oblivious dynamic dictionaries with update/query tradeoffs. In: Proceedings of the Twenty-First Annual ACM–SIAM Symposium on Discrete Algorithms, pp. 1448–1456. SIAM (2010)
7. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4 (2008)
8. Claise, B., Bryant, S.: Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. Technical Report, RFC 5101, January (2008)
9. Dean, J., Ghemawat, S.: LevelDB: a fast and lightweight key/-value database library by Google (2021). http://code.google.com/p/leveldb/
10. DHS Cybersecurity and Infrastructure Security Agency (CISA): Einstein (2021). https://www.cisa.gov/einstein
11. Facebook: RocksDB (2021). https://github.com/facebook/rocksdb
12. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
13. Matsunobu, Y., Dong, S., Lee, H.: MyRocks: LSM-tree database storage engine serving Facebook's social graph. Proc. VLDB Endow. **13**(12), 3217–3230 (2020)
14. NetFlow, Cisco Systems Inc: Netflow Services Solutions Guide. Cisco System, Inc. (2007). http://www.cisco.com/en/US/docs/ios/solutions_docs/netflow/nfwhite.html
15. O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The log-structured merge-tree (LSM-tree). Acta Inform. **33**(4), 351–385 (1996)
16. Pandey, P., Singh, S., Bender, M.A., Berry, J.W., Farach-Colton, M., Johnson, R., Kroeger, T.M., Phillips, C.A.: Timely reporting of heavy hitters using external memory. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 1431–1446 (2020)
17. Paxson, V.: Bro: a system for detecting network intruders in real-time. Comput. Netw. **31**(23), 2435–2463 (1999)
18. Sarkar, S., Papon, T.I., Staratzis, D., Athanassoulis, M.: Lethe: a tunable delete-aware LSM engine. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 893–908 (2020)

Janet Vorobyeva is a Systems and Algorithms Researcher at Sandia National Laboratories. She recently finished a Masters in Computer Science at Stony Brook University.

Daniel R. Delayo is a Graduating Senior in Computer Science at Stony Brook University. His research interests include algorithms and security.

Michael A. Bender is the David R. Smith Leading Scholar of Computer Science at Stony Brook University. Bender was Founder and Chief Scientist at Tokutek, Inc., an enterprise database company, which was acquired by Percona in 2015. Bender works on pure and applied algorithms in I/O-efficient storage systems, parallel computing, and scheduling. He has won several awards, including an R&D 100 Award, a Test-of-Time Award, two Best Paper Awards, and five awards for graduate and undergraduate teaching. He has held Visiting Scientist positions at both MIT and King's College London.

Martin Farach-Colton is a Professor of Computer Science at Rutgers University. He was Founder and CTO at Tokutek, Inc., an enterprise database company, which was acquired by Percona in 2015. Farach-Colton works on pure and applied algorithms in I/O-efficient storage systems, streaming algorithms and string matching. Farach-Colton received his M.D. from Johns Hopkins and his Ph.D. from the University of Maryland. He has been a Member of Technical Staff at Bell Labs (1997–1998) and was an early Employee of Google, Inc. (2000–2002).

Prashant Pandey is a Post Doctoral Researcher at Lawrence Berkeley National Laboratory. He has a PhD in Computer Science from Stony Brook University. His interests include algorithms, storage, and bioinformatics.

Cynthia A. Phillips is a Senior Scientist in the Computing Research Center at Sandia National Laboratories. She received a PhD in Computer Science from MIT. Her research areas include combinatorial optimization, algorithm design and analysis, and parallel computation with more recent work in data science such as streaming algorithms. She is a SIAM Fellow.

Shikha Singh is an Assistant Professor of Computer Science at Williams College. Shikha received her Ph.D. from Stony Brook University in 2018. Her research is in the area of theoretical computer science with a focus on algorithmic game theory and algorithms and data structures for computing on big data. Webpage: http://cs.williams.edu/~shikha/.

Eric D. Thomas has been a Cybersecurity Analyst at Sandia National Laboratories since 1999. His work involves analysis of hacker tactics and techniques, researching and developing mechanisms for thwarting them in the interest of national security. He earned his Master's Degree in Computer Science at the University of California, Davis in 2005.

Thomas M. Kroeger is a Systems and Security Researcher at Sandia National Laboratories. His research interests include security monitoring, secure communications, operating systems, data architectures, and virtualization. He has a PhD in Computer Engineering from UC Santa Cruz. Prior to Sandia he spent 10 years in Silicon Valley working for Network-Alchemy and Validus Medical Systems.