VAST: A Unified Platform for Interactive Network Forensics

Matthias Vallentin, *University of California, Berkeley;* Vern Paxson, *University of California, Berkeley, and International Computer Science Institute;* Robin Sommer, *International Computer Science Institute and Lawrence Berkeley National Laboratory*

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/vallentin

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

VAST: A Unified Platform for Interactive Network Forensics

Matthias Vallentin
vallentin@icir.org
UC Berkeley

Vern Paxson
vern@icir.org
UC Berkeley / ICSI

Robin Sommer
robin@icir.org
ICSI / LBNL

Abstract

Network forensics and incident response play a vital role in site operations, but for large networks can pose daunting difficulties to cope with the ever-growing volume of activity and resulting logs. On the one hand, logging sources can generate tens of thousands of events per second, which a system supporting comprehensive forensics must somehow continually ingest. On the other hand, operators greatly benefit from *interactive* exploration of disparate types of activity when analyzing an incident.

In this paper, we present the design, implementation, and evaluation of VAST (Visibility Across Space and Time), a distributed platform for high-performance network forensics and incident response that provides both continuous ingestion of voluminous event streams and interactive query performance. VAST leverages a native implementation of the actor model to scale both intra-machine across available CPU cores, and inter-machine over a cluster of commodity systems.

1 Introduction

Security incidents often leave network operators scrambling to ferret out answers to key questions: How did the attackers get in? What did they do once inside? Where did they come from? What activity patterns serve as *indicators* reflecting their presence? How do we prevent this attack in the future?

Operators can only answer such questions by drawing upon high-quality logs of past activity recorded over extended time. Incident analysis often starts with a narrow piece of intelligence, typically a local system exhibiting questionable behavior, or a report from another site describing an attack they detected. The analyst then tries to locate the described behavior by examining logs of past activity, often cross-correlating information of different types to build up additional context. Frequently, this process in turn produces new leads to explore iteratively ("peeling the onion"), continuing and expanding until ultimately the analyst converges on as complete of an understanding of the incident as they can extract from the available information.

This process, however, remains manual and time-consuming, as no single storage system efficiently integrates the disparate sources of data (e.g., NIDS, firewalls, NetFlow data, service logs, packet traces) that investigations often involve. While standard SIEM systems such as Splunk aggregate logs from different sources into a single database, their data models omit crucial semantics, and they struggle to scale to the data rates that large-scale environments require.

Based on these needs, and drawing upon our years of experience working closely with operational security staff, we formulate three key goals for a system supporting the forensic process [2]:

Interactivity. The potential damage that an attacker can wreak inside an organization grows quickly as a function of time, making fast detection and containment a vital concern. Further, a system's interactivity greatly affects the productivity of an analyst [16]. We thus desire replies to queries to begin arriving within a second or so.

Scalability. The volume of data to archive and process exceeds the capacity of single-machine deployments. A fundamental challenge lies in devising a distributed architecture that scales with the number of nodes in the system, as well as maximally utilizes the cores available in each node.

Expressiveness. Representing arbitrary activity requires a richly typed data model to avoid losing domain-specific semantics when importing data. Similarly, the system should expose a high-level query language to enable analysts to work within their domain, rather than spending time translating their workflows to lower-level system idiosyncrasies.

In this work, we develop a system for network forensics and incident response that aims to achieve these goals. We present the design and implementation of VAST (*Visibility Across Space and Time*), a unified storage platform that provides: (*i*) an expressive data model to capture descriptions of various forms of activity; (*ii*) the capability to use a single, declarative query language to drive both post-facto analyses and detection of future activity; and (*iii*) the scalability to support archiving and querying of not just log files, but a network's *entire activity*, from high-level IDS alerts to raw packets from the wire.

The key to VAST's power concerns providing the necessary performance to support both very high data volumes (100,000s of events/sec) and interactive queries against extensive historical data. VAST features an entirely asynchronous architecture designed in terms of the actor model [25], a message-passing abstraction for concurrent systems, to fully utilize all available CPU and storage resources, and to transparently scale from single-node to cluster deployments. To support interactive queries, VAST relies extensively on bitmap indexes that we adapt to support its expressive query language.

Our evaluations show that on a single machine VAST can ingest 100 K events/sec for events with 20 fields, reflecting an input rate of 2 M values/sec. Moreover, distributed ingestion allows for spreading the load over numerous system entry points. Users receive a "taste" of their results typically within 1 sec. This first subset helps users to quickly triage the relevance of the result and move on with the analysis by aborting or modifying the current query. We also show that VAST, with its unified approach, can effectively serve as a high-volume packet bulk recorder.

We structure the rest of the paper as follows. In §2 we summarize related work. We present the architecture of VAST in §3 and our implementation in §4. In §5 we evaluate VAST and assess its aptness for the domain. Finally, we conclude in §6.

2 Related Work

Data Warehouses. VAST receives read-only data for archiving, similar to a data warehouse. Dremel [40] stores semi-structured data *in situ* and offers an SQL interface for ad-hoc queries with interactive response. Dremel's query executor forms a tree structure where intermediate nodes aggregate results from their children. VAST generalizes this approach with its actor-based architecture to both data import and query execution.

Succinct [1] also stores data *in situ*, but in compressed flat files that do not require decompression when searched. Internally, Succinct operates on suffix trees and therefore supports point, wildcard, and lexicographical lookup on strings. Other data types (e.g., arithmetic, compound) require transformations into strings to maintain a lexicographical ordering. Succinct exhibits high preprocessing costs and modest sequential throughput, rendering it inapt for high-volume scenarios. When the working set fits in memory, Succinct offers competitive performance, but not when primarily executing off the filesystem.

ElasticSearch [17] is a distributed, document-oriented database built on top of Apache Lucene [36], which provides a full-text inverted index. ElasticSearch hides Lucene behind a RESTful API and a scheme to partition data over a cluster of machines. VAST uses a similar deployment model, but instead provides a semi-structured data model and internally relies on different indexing technology more amenable to hit composition and iterative refinements.

Network Forensics. NET-Fli [20] is a single-machine NetFlow indexer relying on secondary bitmap indexes. The authors also present a promising (though patented) bit vector encoding scheme, COMPAX. Instead of hand-optimizing a system for NetFlow records, VAST offers a generic data model. The separation between base data and indexes has also found application in similar systems [51], with the difference of relying on a column store for the base data instead of a key-value store. The existing systems show how one can design a single-machine architectures, whereas we present a design that transparently scales from single-machine to cluster deployments.

The Time Machine [38] records raw network traffic in PCAP format and builds indexes for a limited set of packet header fields. To cope with large traffic volumes, the Time Machine employs a *cutoff* to cease recording a connection's packets after they exceed a size threshold. The system hard-codes the use of four tree indexes over the connection tuple, and cannot reuse its indexes across restarts. Similarly, NetStore [22], pcapIndex [19], and FloSIS [33] offer custom architectures geared specifically towards flow archival. VAST represents a superset of bulk packet recorders: it supports the same cutoff functionality, and packets simply constitute a particular event type in VAST's data model.

The GRR Rapid Response framework [11] enables live forensic investigations where analysts push out queries to agents running on end-hosts. A NoSQL store accumulates the query results in a central location, but GRR does not feature a persistence component to comprehensively archive end-host activity over long time periods. VAST can serve as a long-term storage backend for host-level data, which allows analysts to query both host and network events in a unified fashion.

Finally, existing aggregators such as Splunk [49] operate on unstructured, high-level logs that lack the semantics to support typed queries, and are not designed for storing data at the massive volumes required by lower-level representations of activity. Splunk in particular cannot dynamically adapt its use of CPU resources to change in workload.

Distributed Computing. The MapReduce [14] execution model enables arbitrary computation, distributed over a cluster of machines. While generic, MapReduce cannot deliver interactive response times on large datasets due to the full data scan performed for each job. Spark [58] overcomes this limitation with a distributed in-memory cluster computing model where data is efficiently shared between stages of computation. However, for rapid response times, the entire dataset must reside preloaded in memory. But analysts can rarely define a working set *a priori*, especially for spatially distant data, which can result in thrashing due to frequent loading and evicting of working sets from memory. We envision VAST going hand-in-hand with Hadoop or Spark, where VAST quickly finds a tractable working set and then hands it off to a system well-suited for more complex analysis.

3 Architecture

To support flexible deployments on large-scale clusters, as well as single machines while retaining a high degree of concurrency, we designed VAST in terms of the actor model [25]. In this model, concurrent entities ("actors") execute independently and in parallel, while providing local fault isolation and recovery. Using unique, location-independent addresses, actors communicate asynchronously solely via message passing. They do not share state, which prevents data races by design.

A related model of computation is communicating sequential processes (CSP) [26] in which processes communicate via synchronous channels. As a result, the sender blocks until the receiver has processed the message. This creates a tighter coupling compared to the asynchronous fire-and-forget semantics of actor messaging. CSP emphasizes the channel while the actor model its endpoints: actors have a location-independent address whereas processes remain anonymous. In the context of distributed systems, the focus on endpoints provides a powerful advantage: the actor model contains a flexible failure propagation model based on monitors, links, and hierarchical supervision trees [4]. These primitives allow for isolating failure domains and implementing local recovery strategies, and thus constitute an essential capability at scale, where component failure is the norm rather than the exception. For these reasons, we deem the actor model a superior fit for our requirements.

We first present the underlying data model and the associated query language (§3.1), and then VAST's components and their structure (§3.2).

3.1 Data Model

Boolean Expression	Symbol
Conjunction	E_1 `&&` E_2
Disjunction	E_1 `||` E_2
Negation / Group	`!` E / `(`E`)`
Predicate	$LHS \circ RHS$
Relational Operator ∘	**Symbol**
Arithmetic	`<, <=, ==, !=, >=, >`
Membership	`in`, `!in`
Extractor (LHS/RHS)	**Semantics**
`:T`	All values having type `T`
`x.y.z`	Value according to schema
`&key`	Event meta data
Types	**Examples**
`bool`	`T`, `F`
`int` / `count` / `real`	`+42 / 42 / -4.2`
`duration` / `time`	`10ms / 2014-09-25`
`string`	`"foo"`, `"b\x2Ar"`
`addr`	`10.0.0.1`, `::1`
`subnet`	`192.168.0.0/24`
`port`	`80/tcp`, `53/udp`, `8/icmp`
`vector<T>` / `set<T>`	`[x, x, x]` / `{x, x, x}`
`table<T,U>`	`{(k,v), ..}`

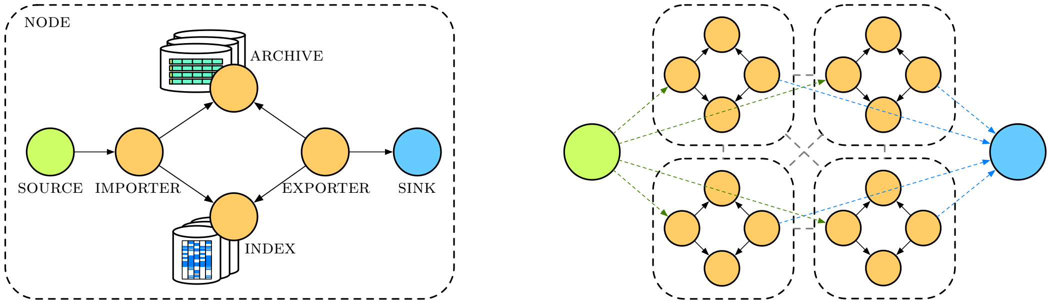
Table 1: VAST's query language.

VAST's data model consists of *types*, which define the physical interpretation of *data*. A type's *signature* includes a type name and type attributes. A *value* combines a type with a data instance. An *event* is a value with additional metadata, such as a timestamp, a unique identifier, and arbitrary key-value pairs. A *schema* describes the access structure of one or more types.

VAST's type system includes *basic types* to represent a single value (booleans, signed/unsigned integers, floating-point, times and durations, strings, IPv4 and IPv6 addresses, subnets, ports), *container types* for bundled values (vectors, sets, tables), and *compound types* to create sequenced structures (records), where each named field holds a value (or `nil` if absent).

Query Language. VAST's query language supports filtering data according to boolean algebra. Table 1 lists the key syntactic elements. A query expression consists of one or more predicates connected with boolean AND/OR/NOT. A *predicate* has the form `LHS ∘ RHS`, with ∘ representing a binary relational operator. VAST supports arithmetic and membership operators. At least one side of the operator typically must be an *extractor*, which specifies the lookup aspect for the value, as follows.

Schema extractors refer to particular values in the schema. For example, in the predicate `http.method == "POST"`, `http.method` is a schema extractor, and `"POST"` is the value to match. *Meta extractors* refer to event metadata, such as `&name` to reference the event name and `&time` the event timestamp. For example, the predicate `&time > now - 1d` selects all events within



(a) Single-machine deployment. The NODE actor accommodates all key system actors.

(b) Cluster deployment. Multiple NODEs peer to form a cluster, which users access transparently.

Figure 1: VAST system architecture.

the last 24 hours. *Type extractors* leverage the strict typing in VAST to perform queries over all values having a given type. For example, the predicate `:addr in 10.0.0.0/8` applies to all IP addresses (any VAST value or record field with type `addr`).

To represent a log file having a fixed number of columns, VAST automatically transforms each line into a `record` whose fields correspond to the columns. VAST enforces type safety over queries and only forwards them to those index partitions with a compatible schema.

3.2 Components

From a high-level view, VAST consists of four key components: (*i*) *import* to parse data from a source into events and assign them a globally unique ID, (*ii*) *archive* to store compressed events and provide a key-value interface for retrieval, (*iii*) *index* to accelerate queries by keeping a partitioned secondary index referencing events in the archive, and (*iv*) *export* to spawn queries and relay them to sinks of various output formats. Each component consists of multiple actors, which can execute all inside the same process, across separate processes on the same host, or on different machines.

A NODE[1] represents a container for other actors. Typically, a NODE maps to a single OS-level process. Users can spawn components all within a NODE (Figure 1(a)) or spread them out over multiple NODEs (Figure 1(b)) to harness available hardware and network resources in the most effective way. NODEs can peer to form a cluster, using Raft [42] for achieving distributed consensus over global state accessible through a key-value store interface similar to etcd [18]. We refer to this globally replicated state as *meta store*. Each NODE has access to its own meta store instance which performs the fault-tolerant distribution of values. A typical cluster deployment exhibits a shared-nothing architecture, where each NODE constitutes a fully independent system by itself. In the following we discuss each of the four components in more detail.

Import. Data enters VAST via SOURCEs, each of which can parse various input formats. SOURCEs produce batches of events and relay them to IMPORTER, where they receive a unique monotone ID. Upon receiving the ID range and assigning them to the events, IMPORTER relays the batch to ARCHIVE and INDEX.

Each event represents a unique description of activity which analysts need to be able to unambiguously reference. This requires each event to have a unique identifier (ID) as meta data independent of its value. The ID also establishes a link between the archive and index component: an index lookup yields a set of IDs, each of which identify a single event in the archive. This yields the following requirements on ID generation: (*i*) *64 bits* to represent a sufficiently large number of events, but not larger since modern processors efficiently operate on 64-bit integers, (*ii*) *monotonicity* because the indexes we use are append-only data structures, and (*iii*) ID generation should also work in *distributed setups*.

The sequentiality requirement precludes approaches involving randomness, such as universally unique identifiers [32]. In fact, any random ID generation algorithm experiences collisions after $\approx \sqrt{N}$ IDs due to the birthday paradox. In combination with the 64-bit requirement, this would degenerate the effective space from 2^{64} to only $\sqrt{2^{64}} = 2^{32}$ IDs. Our approach uses a single distributed counter in the meta store. Requesting a range of N IDs translates to incrementing this counter by N, which returns a pair with the old and new counter value $[o,n)$ denoting the allocated half-open range with $n - o \leq N$ IDs. To avoid high latencies from frequent interaction with the meta store, IMPORTER keeps a local cache of IDs and only replenishes it when running low of IDs.

Archive. The ARCHIVE receives events from IMPORTER and stores them compressed on the filesystem.[2] To avoid frequent I/O operations for small amounts of

[1] We refer to particular actors in a SMALL CAPS font style.

[2] VAST supports LZ4 [37] and Snappy [48] for compression; both trade higher speeds for lower compression ratios.

data, ARCHIVE keeps event batches in a fixed-size memory buffer (by default 128 MB) before writing them to the filesystem. The buffer (which we term *segment*) keeps batches sorted by the ID of their first event. Because events have continuous IDs within a batch, this process ensures strictly monotonic IDs within a segment.

ARCHIVE exposes a key-value interface: queried with a specific ID, it returns a batch containing the ID. The alternative, returning the single matching event, only works for small requests, but would quickly bottleneck the messaging subsystem for moderate request volumes. Internally, ARCHIVE operates at the granularity of segments to further group event batches into larger blocks suitable for sequential filesystem I/O. ARCHIVE keeps an LRU cache of a configurable number of segments in memory. In the future, we plan to store data in a format also shareable with other applications, e.g., HDFS [46].

Index. By itself, ARCHIVE does not provide efficient access to data, since extracting events with specific properties would require a full scan of the archive. Therefore, VAST maintains a comprehensive set of secondary indexes, which we divide up in horizontal partitions as a unit of data scaling.

We chose bitmap indexes [41] because they provide an excellent fit for the domain. First, appending new values only requires time linear in the number of values in the new data, which is optimal and yields deterministic performance. Second, bitmap indexes have space-efficient representations that enable us to carry out bitwise operations without expanding them. Third, bitmap indexes *compose* efficiently: intermediate results have the form of bit vectors and combining them with logical operation translates into inexpensive bitwise operations. In §4.2 we describe bitmap indexes in more detail.

VAST's index consists of horizontal PARTITIONs, each of which manages an independent set of bitmap indexes. An *active* partition is mutable and can incorporate events, whereas a *passive* partition is an immutable unit which the scheduler manages during query processing. INDEX relays each arriving batch of events to the currently active PARTITION, which spawns a set of INDEXERs, one per event type. An INDEXER may internally further helper actors, e.g., record INDEXERs use one helper per field. In comparison to a relational database, this architecture provides a fine-grained concurrent equivalent to tables (INDEXERS) and their attributes. PARTITION relays each batch concurrently to *all* INDEXERs, each of which processes only the subset matching its type. The copy-on-write message passing semantics of the actor model implementation makes this an efficient operation (INDEXERs only need read-access to the events) while providing a high degree of parallelism.

Export. While the import component handles event ingestion, the export component deals with retrieving

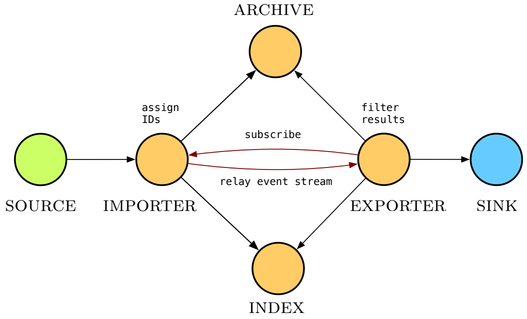


Figure 2: Continuous query architecture.

events through queries. There exists one EXPORTER per query, who sends its result to SINKs for rendering. VAST currently includes ASCII, JSON, PCAP [35], Bro [43], and Kafka [29] SINKs.

For *historical queries* over existing data, INDEX analyzes the abstract syntax tree (AST) of the query expression to determine the relevant PARTITIONs and constructs a schedule to process them sequentially. This process begins with *partition pruning*: the selection of relevant partitions for a query. To this end, VAST uses a "meta index," consisting of event meta data and event type information. The meta index has different requirements than the event indexes within a partition: it must tolerate immense throughput and update rates. We currently associate with each partition the time period its events spans and record the entire partition schema.

After pruning, INDEX relays the query to the remaining PARTITIONs, which then deconstructs the AST into its predicates to match them with INDEXERs. If necessary, PARTITION loads the INDEXER from the filesystem into memory. Upon performing the predicate lookup, INDEXERs send their hits back to their PARTITION, where they trigger a re-evaluation of the query expression. If the evaluation yields a change, PARTITION relays the *delta hits* from the last evaluation up to INDEX, which in turn forwards them to the subscribed EXPORTERs. As soon as the first hits arrive at an EXPORTER, extracting events can begin by asking ARCHIVE. When EXPORTER receives an answer in the form of a batch of events, it concurrently prefetches another batch proceeding with the "candidate check" to filter out false positives, which may be necessary for some indexes types (e.g., when using binning for floating point values, see §4.2). Finally, EXPORTER sends the matching results back to SINK. The process terminates after EXPORTER has no more unprocessed hits to extract.

VAST also supports *continuous queries* to subscribe to new results as they arrive. As we illustrate in Figure 2, EXPORTER can subscribe to a copy of the full incoming event feed at IMPORTER to filter out those events matching the query expression. Since IMPORTER and EXPORTER live in the same process this operation does not copy any data. In fact, a continuous query is effectively a candidate check, with the only difference that events now come from IMPORTER instead of ARCHIVE. The main challenge lies in efficiently performing this check. To this end, EXPORTER derives from the AST of the query expression a visitor performing the candidate check. EXPORTER constructs one visitor per unique event type on the fly, and dispatches to the relevant visitor efficiently through a hash table. For example, an expression may include the predicate `:addr in 10.1.1.0/24`. If the current event has no data of type `addr`, the visitor discards the predicate.

3.3 Distribution

When the amount of data exceeds the capacity of a single machine, VAST can distribute the load over multiple machines, as we show in Figure 1(b) for 4 peering NODEs.

Cluster Deployment. In case a SOURCE produces events at a rate that overloads a single system, it can load-balance its events over all NODEs. Alternatively, users can pin SOURCEs to a specific set of NODEs (e.g., if certain data should land on a more powerful system). The dual occurs during querying: the user decides on which NODEs to spawn an EXPORTER, each of which will relay its events to the same SINK. In the common case of round-robin load-balancing of input data, a user query results in spawning one EXPORTER on all NODEs.

Fault Tolerance. Coping with NODE failure concerns two aspects: data loss and query execution. To avoid permanent data loss, we assume a reliable, distributed filesystem, e.g., HDFS. Data loss can still occur during ingestion, when ARCHIVE and INDEX have not yet written their data to the filesystem. VAST minimizes this risk by writing data out as quickly as possible. When a NODE fails during query execution, another takes over responsibility of its persistent ARCHIVE and INDEX data, and re-executes the query on its behalf. EXPORTER can periodically checkpoint its state (consisting of index hits and event identifiers of results that have passed the candidate check) to reduce the amount of duplicate results.

4 Implementation

We now present some implementation highlights of the previously discussed architecture with a focus on actors (§4.1), bitmap indexes (§4.2), and queries (§4.3).

4.1 Actors

We implemented the components discussed in §3.2 with the C++ Actor Framework (CAF), a native implementation of the actor model [10].

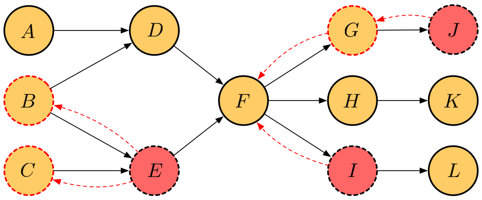


Figure 3: Flow-control implemented as back-pressure: overloaded nodes J, I, and E propagate their load status upstream such that data sources can throttle their sending rate.

Performance. CAF offers a high-performance runtime with a type-safe messaging layer that exhibits minimal memory overhead. The runtime can distribute actors dynamically, within the same process, across the network, or to GPUs. Within the same process, CAF uses copy-on-write semantics to enable efficient messaging. CAF's networking layer handles actor communication and routing *transparent* to the user: the runtime decides whether sending a message translates into a local enqueue operation into the recipient's mailbox, or whether a middleman serializes the message and ships it across the network.

CAF's copy-on-write messaging proves particularly valuable during the indexing process, where PARTITION sends the *same set of events* to each INDEXER without incurring a copy of the data. Although each INDEXER sees the full data feed, this method still runs faster than chopping up the input sequence and incurring extra memory allocations. Such a computation style resembles GPU programming where we make available data "globally" and each execution context picks its relevant subset.

Flow Control. CAF operates entirely asynchronously and does not block: immediately after sending a message an actor can dequeue a message from its mailbox. This makes it easy for data producers to overwhelm downstream nodes if not equipped with enough processing capacity. A naive reaction entails provisioning more buffer capacity at the edge so that the system can receive more messages. But without including the true bottlenecks in the decision, buffer bloat [31] only worsens the situation by introducing higher latency and jitter. Flow control attempts to prevent this scenario from happening: *back-pressure* signals overload back to the sender, *load shedding* reduces the accumulating tasks at the bottleneck, and *timeouts* at various stages in the data flow graph bound the maximum response time.

CAF currently does not support flow control. During data import, data producers (SOURCEs) can easily overload downstream components (ARCHIVE and INDEX). To throttle the sending rate of SOURCEs, we implemented a simple back-pressure mechanism: when an actor becomes overloaded, it sends an overload message to all its registered upstream components, which either propagate it further, or if being the data producer themselves, throttle their sending rate (see Figure 3). When an overloaded actor becomes underloaded again, it sends an underload message to signal upstream senders that it can process data again. This basic mechanism works well to prevent system crashes due to overloads, but does not help at absorb peak input rates. To prevent data loss of non-critical, latency-insensitive events, queueing brokers that spill to the filesystem (e.g., Kafka [29]) at the edges help smoothing the data arrival rate to facilitate resource provisioning for the average case. We are currently working with the CAF developers to integrate various forms of flow control deeper into the runtime.

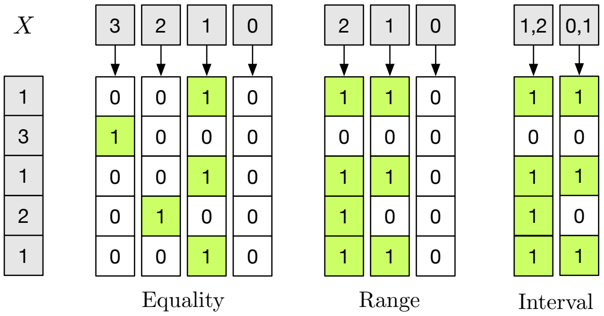
4.2 Composable Indexing

VAST exclusively relies on bitmap indexes to accelerate queries. We begin in §4.2.1 with briefly summarizing existing work, which we then rely on in §4.2.2 to define composable higher-level index types.

4.2.1 A Unified Indexing Framework

Traditional tree and hash indexes [6, 30] provide a quick entry point into base data, but they do not compose efficiently for higher-dimensional search queries. Inverted and bitmap indexes avoid this problem by adding an extra level of indirection: instead of looking up base data directly, they operate on the IDs of the base data, allowing for combining results from multiple index lookups via set operations. Consider the event values $x^{(\alpha)}$ where x represents a numeric value and α its ID, e.g., $1^{(0)}$, $3^{(1)}$, $1^{(2)}$, $2^{(3)}$, and $1^{(4)}$. An inverted index represents the events as a mapping from values to *position lists*: $1 \rightarrow \{0,2,4\}$, $2 \rightarrow \{3\}$, and $3 \rightarrow \{1\}$. A bitmap index is an isomorphic data structure where the position lists have the form of *bit vectors*:[3] $1 \rightarrow \langle 10101 \rangle$, $2 \rightarrow \langle 00010 \rangle$, and $3 \rightarrow \langle 01000 \rangle$. When the distinction does not matter, we use the term *identifier set* to mean either position list or bit vector.

There exist hybrid schemes which combine both index types in a single data structure [7], but VAST currently implements its algorithms only in terms of bitmap indexes, where operations from boolean algebra (set, intersection, complement) naturally map to native CPU instructions. In general, an index I provides two basic primitives: adding new values and looking up existing ones under a given predicate. To add a new value $x^{(\alpha)}$, the index adds α to the identifier set for x. A lookup under a predicate $I \circ x$ retrieves the identifier set $S = \{\alpha \mid x^{(\alpha)} \in I\}$. The *size* of and index $|I| = N$ represents the number of values entered and the *cardinality* $\#I = C$ the number of distinct values.



X	3	2	1	0	2	1	0	1,2	0,1
1	0	0	1	0	1	1	0	1	1
3	1	0	0	0	0	0	0	0	0
1	0	0	1	0	1	1	0	1	1
2	0	1	0	0	1	0	0	1	0
1	0	0	1	0	1	1	0	1	1

Figure 4: Equality, range, and interval encoding exemplified using bitmap indexes.

Our example has $N = 5$ and $C = 3$. In the following, we sketch key concepts that affect the inherent space-time trade-off during the implementation of indexes, which include binning, coding, compression, and composition.

Binning reduces the cardinality of an index by grouping values into bins or rounding floating-point values to a certain decimal precision. For example, we could create bins $[1,2] \rightarrow \langle 10111 \rangle$ and $[3,4] \rightarrow \langle 01000 \rangle$, which reduces the cardinality of the index to $C = 2$. The surjective nature of binning introduces false positives and therefore requires a *candidate check* with the base data to verify whether a certain hit qualifies as an actual result. A candidate check can easily dominate the entire query execution time due to materializing additional base data (high I/O costs) and extra post-processing. Therefore, choosing an efficient binning strategy requires careful tuning and domain knowledge, or advanced adaptive algorithms [44, 47].

Encoding determines how an index maps values to identifier sets. We show in Figure 4 the three major existing encoding schemes for a fixed cardinality $C = 4$, which can represent values 0–3. *Equality encoding* associates each value with exactly one identifier set. This scheme reflects our running example and consists of exactly C identifier sets. *Range encoding* associates each value $x^{(\alpha)}$ with a range of $C - 1$ identifier sets such that an ID α lands in i sets where $x \leq i$. We can omit the last identifier set because $x \leq C - 1$ holds true for all possible values. *Interval encoding* splits the index into $\lceil \frac{C}{2} \rceil$ overlapping slices, each of which covers half of the values. In our example, we have two the intervals $[0,1]$ and $[1,2]$.

Compression algorithms for bit vectors typically use variations of run-length encoding, which support bitwise operations without prior decompression. There exist numerous algorithms: BBC [3], WAH [56], COMPAX [20], CONCISE [12], WBC/EWAH [57, 34], PLWAH [15], DFWAH [45], PWAH [53], VLC [13], and VAL [24]. We chose EWAH for VAST because when we began our project software patents covered (and still do) the other attractive candidates (WAH, PLWAH, COMPAX), which would have prevented us from releasing our project as

[3] The literature often uses the term "bitmap" to refer to a bit vector, i.e., a sequence of bits. We use "bit vector" instead to avoid confusion between "bitmap" and "bitmap index."

open-source software. EWAH also trades space for time: while exhibiting a slightly larger footprint, it executes faster in certain conditions [23] because it can skip entire blocks of words.

Multi-component indexes combine several individual index instances (which might use different approaches) such that each covers a disjoint partition of the value domain. Doing so provides an exponential reduction in space by decreasing the size of the value domain by a multiplicative factor for each component. We can decompose a value x into k components $\langle x_k, \dots, x_1 \rangle$ by representing it with respect to a fixed *base* (or radix) $\beta = \langle \beta_k, \dots, \beta_1 \rangle$: $x = \sum_{i=1}^{k} x_i \beta_i$, where $x_i = \lfloor x / \prod_{j=1}^{i-1} \beta_j \rfloor \bmod \beta_i$, for all $i \in \{1, \dots, k\}$. This decomposition scheme directly applies to the index structure as well: a multi-component index $K^\beta = \langle I_k, \dots, I_1 \rangle$ consists of k indexes, where each I_i covers a total of β_i values. A base is *uniform* if $\beta_i = \beta_j$ for all $i \neq j$. A uniform base with $\beta_i = 2$ for all $1 \leq i \leq k$ yields the *bit-sliced index* [55], because each x_i can only take on values 0 and 1. We denote this special case by $\Theta^k = K^\beta$ where $|\beta| = k$ and $\beta_i = \beta_j = 2$ for all $i \neq j$. Further, we define $\Phi^w = K^\beta$ where $\prod_{i=1}^{k} \beta_i \leq 2^w$ as an index which supports up to 2^w values.

For example, consider a two-component index $K^\beta = \langle I_2, I_1 \rangle$ with $\beta = \langle 10, 10 \rangle$, which supports 100 distinct values. Appending a value $x^{(\alpha)} = 42$ involves first decomposing it into $\langle 4, 2 \rangle$, and then appending $4^{(\alpha)}$ to I_2 and $2^{(\alpha)}$ to I_1. Looking up the value $x = 23$ begins with decomposing x into $\langle 2, 3 \rangle$, and then proceeds with computing $I_2 = 2 \wedge I_1 = 3$. The final step resolves each component lookup according to its encoding scheme. Operators other than $\{=, \neq\}$ require more elaborate lookup algorithms [8, 9], which we lay out in greater detail separately [52].

4.2.2 Higher-Level Indexing

For each type in VAST's data model, there exist different requirements derived from the desired query operations. For example, numeric values commonly involve inequality comparisons and IP address lookups top-k prefix search. Different lookup operations require different index layouts.

Per §3.1, a *value* consists of a *type* and corresponding *data*. A value can exhibit no data, in which case it only carries type information. We define the *value index* $\mathbb{V} = \langle N, D \rangle$ as a composite structure with a *null index* N to represent whether data is null (implemented as single identifier set), and a *data index* D to represent a type-specific index, whose instantiations we describe next.

Integral Indexes. The *boolean index* $\mathbb{B} = \langle S \rangle$ for type `bool` consists of a single identifier set, where $\alpha \in S$ implies $x^{(\alpha)} =$ `true`.

For types `count` and `int`, the challenge lies in both supporting lookups under $\{<, \leq, =, \neq, \geq, >\}$, as well as representing 2^{64} distinct values. To address the high cardinality challenge, we use a multi-component index Φ^{64}. We found that a uniform base 10 works well in practice. To support the desired arithmetic operations, we use range coding, which supports both equality and inequality lookups efficiently.

Signed integers introduce a complication: we cannot map negative values to array indices during encoding, and only positive numbers work with value decomposition. For a w-bit signed integer, we therefore introduce a "bias" of 2^{w-1}, which shifts the smallest value of -2^{w-1} to 0 in the unsigned representation. This allows us to use the same index type for signed an unsigned integers internally. Thus, we define the *count index* as $\mathbb{C} = \Phi^{64}$ and the *integer index* as $\mathbb{I} = \Phi^{64}$ with the aforementioned bias.

Floating-Point Index. Type `real` corresponds to a IEEE 754 double precision floating point value [28], which consists of one sign bit, 11 bits for the exponent, and 52 bits for the significand. Consequently, we construct the *real index* $\mathbb{F} = \langle S, E, M \rangle$ as a boolean index $S = \mathbb{B}$ for the sign, a bit-sliced index $E = \Theta^{11}$ for the exponent, and a bit-sliced index $M = \Theta^{52}$ for the significand. Varying the number of bits in E and M allows for trading space for precision without the need to round to a specific decimal point.

Temporal Indexes. VAST represents `duration` data as a 64-bit signed integers in nanosecond resolution, which can represent ±292.3 years. Since `duration` and `int` are representationally equal, the *duration index* $\mathbb{D} = \mathbb{I}$ directly maps to an integer index.

The type `time` describes a fixed point in time, which VAST internally expresses as `duration` relative to the UNIX epoch. Thus, the *time index* $\mathbb{T} = \mathbb{D}$ is representationally equal to the duration index.

String Index. Existing string indexes rely on a dictionary to map each unique string value to a unique numeric value [50]. However, constructing a space-efficient dictionary poses a challenge in itself [39, 27]. Moreover, this design only supports equality lookups naturally: for substring search, one must search the dictionary key space first to get the identifier sets, and then perform the lookup for each identifier set, and finally combine the result. Instead of using a stateful dictionary, one can rely on hashing to compute a unique string identifier [51]. The possibility of collisions now requires a candidate check. While space-optimal due to the absence of a dictionary, and time-efficient due to fast computation, this approach does not support substring search.

We propose a new approach for string indexing that supports both equality and substring search, yet operates in a stateless fashion without dictionary. Our *string index* $\mathbb{S} = \langle \phi, \kappa_1, \dots, \kappa_M \rangle$ consists of an index $\phi = K^\beta$ for the string length, plus M indexes $\kappa_i = \Phi^8$ per character where

M is the largest string added to the index. When representing the character-level indexes κ_i as a bit-sliced index Θ^8, we obtain efficient case-insensitive (substring) search for ASCII-encoded strings. This works because only the 6th bit determines casing in ASCII, and by simply omitting the corresponding identifier set Θ_6^8 during lookup, case-insensitive search executes *faster* than case-sensitive search. We plan on supporting search via a subset of regular expressions by compiling a pattern into an automaton performing a sequence of per-character lookups. Likewise this structure lends itself to similarity search, e.g., via edit-distance.

This design works well for bounded, non-uniform string data, such as URLs or HTTP user agents. For other workloads we fall back to hashing in combination with *tokenization*. This preprocessing step splits a string according to a pattern (e.g., whitespace for text, '/' for URIs, etc.), which then creates a set of multiple smaller strings. Adaptively switching between the index types presents an interesting opportunity for future work, e.g., by inspecting both the nature of user queries as well as inferring the data distribution.

Network Indexes. IP addresses constitute a central data type to describe endpoints of communicating entities. The most common operation on IP addresses consists of top-k prefix search, e.g., $I \in$ `192.168.0.0/24` or $I \notin$ `fd00::/8`. We can consider equality lookup as a special case when $k = 32$ and $k = 128$ for IPv4 and IPv6, respectively. There exists a standardized scheme to embed a 32-bit IPv4 address inside a 128-bit IPv6 address [5]: set the first 96 bits to 0 and copy the IPv4 address in the last 32 bits. This yields the *address index* $\mathbb{A} = \Theta^{128}$ where each bit in the address corresponds to one identifier set in the index.

The `subnet` type consists of a network address and a prefix. Typical queries involve point lookups of IP addresses (e.g., `192.168.0.42` $\in I$), and subset relationships to test whether one subnet contains another (e.g., `192.168.0.0/28` $\subseteq I$). The *subnet index* $\mathbb{U} = \langle \mathbb{A}, P \rangle$ consists of an address index $\mathbb{A}$ and a single equality-encoded index P for the prefix.

The `port` type consists of a 16-bit number and a transport protocol type. The *port index* $\mathbb{P} = \langle \Phi^{16}, T \rangle$ consists of an index for the 16-bit bit port number and a single equality-encoded index T for the different port types.

Container Indexes. Container types include `vector`, `set`, and `table`. A container contains a variable number of elements of a homogeneous type, unlike `records`, which allows for fixed-length heterogeneous data with named fields. For example, a `set` describes DNS lookup results, where a single host name has associated multiple A records. We support cardinality and subset queries on containers.

The design of string indexes generalizes to containers. Let M denote the maximum number of elements in a container. We define the *vector index* $\mathbb{X}^V$ and *set index* $\mathbb{X}^S$ both as $\langle \phi, \mathbb{V}_1, \ldots, \mathbb{V}_M \rangle$: an index $\phi = K^\beta$ for the container size and M value indexes $\mathbb{V}_i$. The *table index* $\mathbb{X}^T = \langle \phi, X_1, \ldots, X_M, Y_1, \ldots, Y_M \rangle$ consists of an index for the size ϕ, a sequence of value indexes $X_i = \mathbb{V}$ for the table keys, and a sequence of value indexes $Y_i = \mathbb{V}$ for the table values. Tables support key lookup, value lookup, and checking for specific mappings.

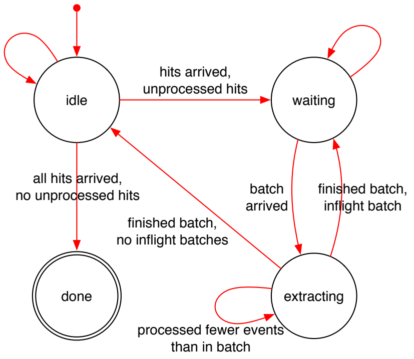

idle
hits arrived, unprocessed hits
waiting
all hits arrived, no unprocessed hits
batch arrived
finished batch, inflight batch
finished batch, no inflight batches
done
extracting
processed fewer events than in batch

Figure 5: The QUERY state machine.

4.3 Queries

The actor model provides an apt vehicle to implement a fully asynchronous architecture, which enables VAST to deliver interactive response times. We illustrate how this applies to query processing in the following.

Finite state machines. We found that finite state machines (FSMs) prove an indispensable mechanism to ensure correct message handling during query execution. Recall that NODE spawns an EXPORTER for each query to bridge ARCHIVE and INDEX. We implemented EXPORTER as a finite state machine (see Figure 5), which begins in `idle` state. Upon receiving new hits EXPORTER asks ARCHIVE for the corresponding batches and transitions into `waiting`. As soon as the first batch arrives, it transitions into `extracting`, from where a user can selectively control it to fetch specific results. By letting the user drive the extraction, VAST does not consume resources unless needed.

Predicate-level caching. To speed-up related queries, INDEX maintains a predicate cache. If hits for the expression `A || B` exists already, then a new expression `A && D` only requires looking up `D`. This makes iterative query styles viable, where the analyst keeps on refining a filter until having pin-pointed the desired information.

Evaluating expressions. When INDEX receives a query consisting of multiple predicates, VAST evaluates them concurrently. Recall from §3.2 that during a historical query INDEXERs send their hits back to PARTITION, where they trigger an evaluation of the expression. If the evaluation yields new hits (i.e., a bit vector with new

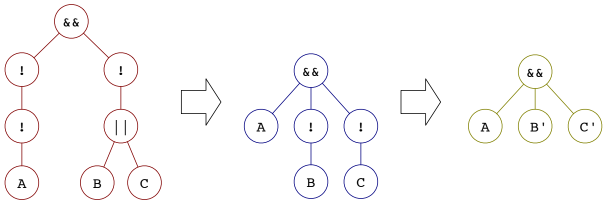


Figure 6: Expression normalization: negation normal form (NNF) and negation absorbing. The expression $\overline{\overline{A}} \wedge \overline{B \vee C}$ first becomes $A \wedge \overline{B} \wedge \overline{C}$. Thereafter, we can absorb negations further and reduce the intermediate expression to $A \wedge B' \wedge C'$, e.g., if $B = \overline{I < x}$, then $B' = I \geq x$.)

1-bits), PARTITION forwards them to INDEX, which in turn relays them to EXPORTER.

To minimize latency and relay hits as soon as possible, we normalize queries to negation normal form (NNF), which eliminates double-negations and pushes negations inwards to the predicate level. We also absorb remaining negations into the predicates, which is possible because each operator has a complement (e.g., $<$ and $\geq$). Figure 6 illustrates the normalization procedure. The absence of negations, aside from saving an extra complement operation, has a useful property: a 1-bit will never turn to 0 during evaluation.

To understand this benefit, consider a predicate A which decomposes into n sub-predicates. This may occur for predicates of the form `:addr in 172.16.0.0/16`, where the type extractor acts as a placeholder resolving to n concrete schema extractors. When PARTITION sends A to the n INDEXERS, they report their hits asynchronously as soon as they become available. PARTITION continuously re-evaluates the AST for new arriving H_i, until having computed $A = H_1 \vee \cdots \vee H_n$. As soon as a re-evaluation yields one or more new 1-bits, PARTITION relays this delta upstream to INDEX. If we kept the negation $\overline{A}$, we would to wait for *all n* hits to arrive in order to ensure we are not producing false positives. Without negations, we can relay this change immediately since a 1-bit cannot turn 0 again in a disjunction.

5 Evaluation

We evaluate our implementation in terms of throughput (§5.1), latency (§5.2), and storage requirements (§5.4). We performed measurements with two types of inputs: synthetic workloads that we can precisely control, and real-world network traffic. For the former, we implemented a benchmark SOURCE that generates input for VAST according to a configuration file. The SOURCE generates all synthetic data in memory to avoid adding I/O load. For real-world input, we use logs from Bro and raw PCAP traces. For the latter, VAST functions as a flow-oriented bulk packet recorder.

VAST comprises 36,800 lines of C++14 code (excluding whitespace and comments), plus 6,700 line of C++ unit tests verifying the system's building blocks and basic interactions. We expand on these checks with an end-to-end test of whether the entire pipeline—from import, over querying, to export—yields correct results. For validation, we processed our ground truth (Bro logs and PCAP traces) separately and cross-checked against the query results VAST delivers. We found full agreement.

We conducted our single-machine evaluation experiments on a 64-bit FreeBSD system with two 8-core Intel Xeon E5-2650 CPUs with 128 GB of RAM and four 3 TB SAS 7.2 K disks (RAID 10 with 2 GB of cache). Our dataset encompasses a full-packet trace from the upstream link at the International Computer Science Institute (ICSI), containing 10 M packets over a 24-hour window on Feb. 24, 2015. We further use 3.4 M Bro connection logs derived from this trace. For our cluster experiments, we use 1.24 B Bro connection logs (152 GB), split into N slices for N worker nodes, with N ranging from 1 to 24. Each worker node runs FreeBSD 10 on a system with two 8-core Intel Xeon E5430 CPUs with 12 GB of RAM and 2 x 500 MB SATA disks. An additional machine with two Xeon X5570 CPUs and 24 GB performs the slicing. The machines share a 1 GE network link.

5.1 Throughput

One key performance metric represents the rate of events that VAST can ingest. Recall the data flow: SOURCEs parse and send input to a system entry point, an IMPORTER, which dispatches the events to ARCHIVE and INDEX. Because we can spawn multiple SOURCEs for arbitrary subsets of the data, we did not optimize SOURCEs at this stage in the development, nor ARCHIVE since it merely sequentially compresses events into fixed-size chunks and writes them out to the filesystem. Instead, we concerned ourselves with achieving high performance at the bottleneck: INDEX, which performs the CPU-intensive task of building bitmap indexes.

Macro Benchmark. For the ingestion benchmark, we configured a batch size of 65,536 events at SOURCE, after observing that greater values entail slightly poorer performance and higher variance (we tested up to 524,288 events per batch).

Figure 7 shows the event rates for three data formats (Bro, PCAP, and a benchmark test) at SOURCE, ARCHIVE, and INDEX as a function of number of cores provided to CAF's scheduler in the single system setup. The y-axis shows the throughput in events per second; note the log scale. As mentioned above, by design SOURCE and ARCHIVE exhibit a fairly constant throughput rate. The highly concurrent architecture complicates measurements of aggregate throughput at INDEX, because there

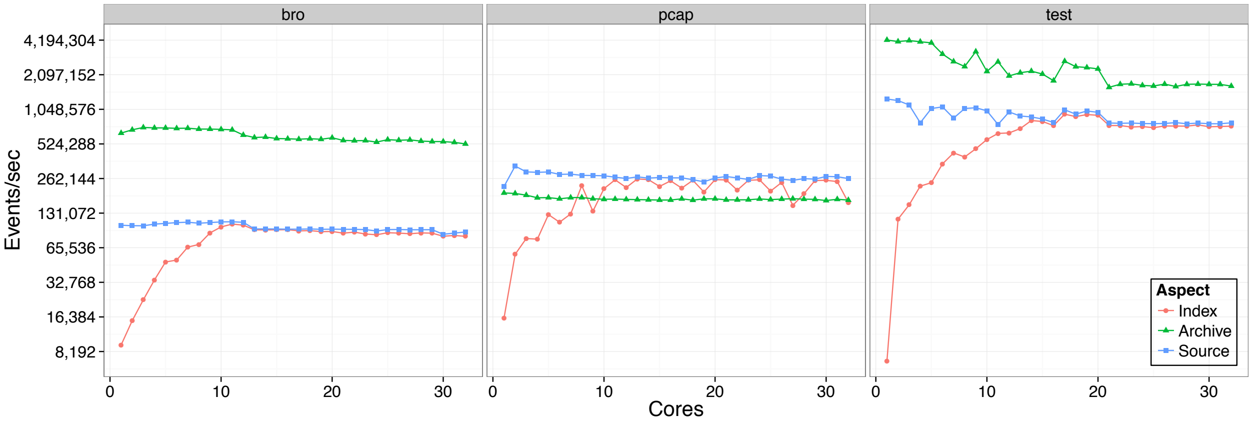


Figure 7: Throughput in events per second as a function of the number of cores for three data types: Bro connection logs, PCAP traces, and a test source generating synthetic events.

Label	Results	Query	Description
A	374	resp_h == 2001:7fe::53	Connections to a specific IPv6 address
B	942	(duration > 1000s || resp_bytes > 40000) && service == "dns"	Anomalous DNS / zone transfers
C	13	orig_h in 192.150.186.0/23 && orig_bytes > 10000 && service == "http"	Outgoing HTTP requests > 10 KB (exfiltration)
D	3	duration > 1h && service == "ssh"	Long-lived SSH sessions
E	969,092	conn_state != "SF"	TCP sessions lacking normal termination
F	4812	:addr in 192.150.186.0/23 && :port == 3389/?	All RDP involving ICSI connections
G	1,077	:addr in 192.150.186.0/23 && :port == 3389/?	Same as above, but applied to PCAP trace
H	34	&time > 2015-02-04+10:00:00 && &time < 2015-02-04+11:00:00 && ((src == 77.255.19.163 && dst == 192.150.187.43 && sport == 49613/? && dport == 443/?) || (src == 192.150.187.43 && dst == 77.255.19.163 && sport == 443/? && dport == 49613/?))	Extract all packets from a single connection specified by its 4-tuple and restricted to a one-hour time window
I	187,015	&time > 2015-02-04+10:00:00 && &time < 2015-02-04+11:00:00 && :addr == 192.150.187.43	All traffic from a single machine within a one-hour window

Table 2: Test queries for single-machine throughput and latency evaluation. The top 6 queries run over Bro connection logs and the bottom 3 over a PCAP trace.

exist multiple running INDEXERs, but not all start and finish at the same time. Therefore we compute throughput at INDEX as the number of events processed between start and end of the measurement. Consequently, the throughput can never exceed the input rate of SOURCE. We observe that the indexing rate approaches the input rate for all sources at around 10 cores. Giving CAF's work-stealing scheduler, more cores yield no further improvement; in fact, performance decreases slightly. We presume this occurs to due thrashing since CAF does not pin the worker threads to a specific core, which increases context switches and cache evictions.

VAST parses Bro events at a rate of roughly 100 K events per second, with each event consisting of 20 different values, yielding an aggregate throughput of 2 M values per second. For PCAP, events consist of only the 4-tuple plus the full packet payload; the latter do not need indexing. VAST can read at around 260 K packets per second with libpcap. Since ARCHIVE does not skip the payload, it cannot keep up with the input rate. This suggests that we need to parallelize this component in the future, which can involve spawning one COMPRESSOR per event batch to parallelize the process. With our test SOURCE, INDEX converges to the input rate at around 14 cores, and we observe input rates close to 1 M events per second. We conclude that VAST meets the performance and scalability goals for data import on a single machine: the system scales up to the point of the input rate after 10-14 cores.

Micro Benchmark. To better understand where VAST spends its time, we instrumented CAF's scheduler to get fine-grained, per-actor resource statistics. This involved bracketing the job execution with resource tracking calls (getrusage), i.e., we only measure actor execution and leave CAF runtime overhead, mostly out of the picture.

In Figure 8, we plot user versus system CPU time for all key actors. Each point represents a single actor instance, with its size scaled to the utilization: user plus system CPU time divided by wallclock time. Note the log scale on both axes. In the top-right corner, we see ARCHIVE, which spends its time compressing events (user) and writing chunks to disk (system). Likewise, INDEX appears nearby, which manages primarily PARTITIONs and builds small "meta indexes" based on time to quickly identify which PARTITION to consider during a query. The bulk of the processing time spreads over numerous INDEXERs,

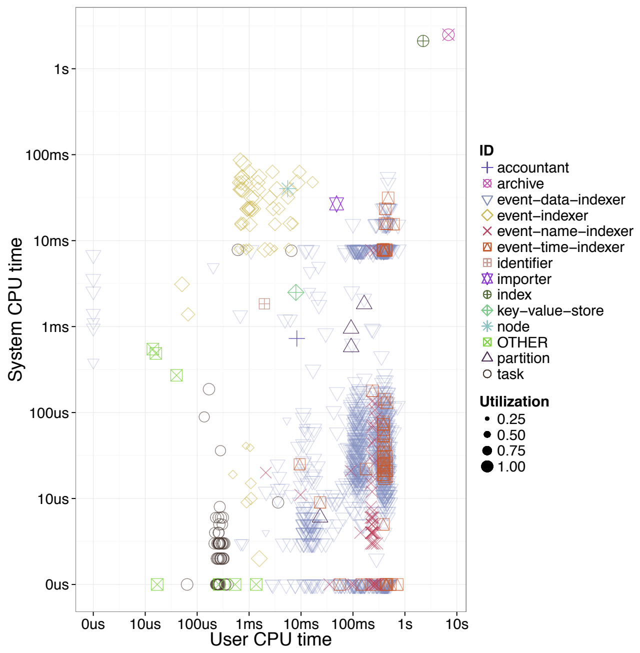


Figure 8: User versus system CPU time for key actors. Each point represents a single actor instance, with point size scaled to utilization: user plus system CPU time divided by wallclock time.

which we can see accumulating on the right-hand side, because building bitmap indexes is a CPU-bound task.

5.2 Latency

Query response time plays a crucial role in assessing the system's viability. VAST spawns one EXPORTER per query, which acts as a middleman receiving hits from INDEX and retrieving the corresponding compressed chunks of events from ARCHIVE. This architecture exhibits two interleaving latency elements: the time (*i*) from the first to the last set of hits received from INDEX, and (*ii*) from the first to the last result sent to a SINK after a successful candidate check.

To evaluate these latency components, we use the set of test queries given in Table 2, which a security operator for a large enterprise confirmed indeed reflect common searches during an investigation.

Query Pipeline. Figure 9 illustrates the latency elements seen over the test queries. For all queries, we ran VAST with 12 cores and a batch size of 65,536. The first red bar corresponds to the time it took until EXPORTER received the first set of hits from INDEX. The green bar shows the time until EXPORTER has sent the first result to its SINKs. We refer to this as "taste" time, since from the user perspective it represents the first system response. The blue bar shows the time until EXPORTER has sent the full set of results to its SINK. The black transparent

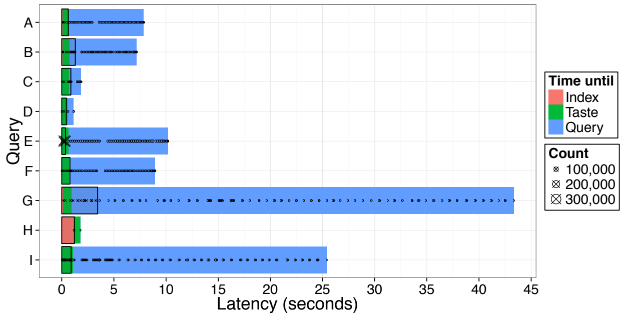


Figure 9: Query pipeline reflecting various stages of single-node execution. The first stage (Index) may appear absent because it can take too little time to manifest in the plot.

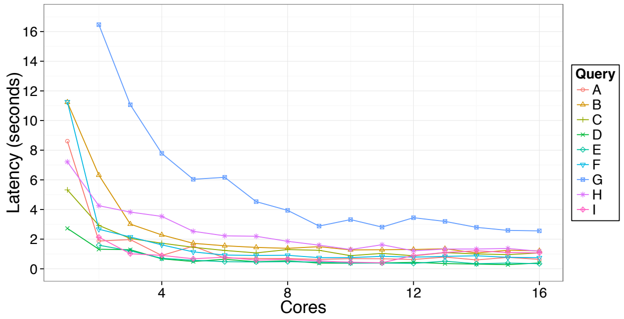


Figure 10: Index latency (full computation of hits) as a function of cores.

box corresponds to the time when INDEX finished the computation of hits. Finally, the crosses inside the bar correspond to points in time when hits arrive, and the circles to the times when EXPORTER finishes extracting results from a batch of events.

We see that extracting results from ARCHIVE (blue bar) accounts for the largest share of execution time. Currently, this time is a linear function of the query selectivity, because EXPORTER does not perform extraction in parallel. We plan to improve this in the future by letting EXPORTER spawn a dedicated helper actor per arriving batch from ARCHIVE, allowing for concurrent sweeps over the candidates. Alternatively, we could offload more computation into ARCHIVE. Selective decompression algorithms [21] present an orthogonal avenue for further improvement.

Index. VAST processes index lookups in a continuous fashion, with first hits trickling in after a few 100 msecs. Figure 10 shows that nearly all index lookups fully complete within 3 seconds once we use more than 4 cores. For query G, we observe scaling gains up to 10 cores. This particular query processes large intermediate bit vectors during the evaluation, which require more time to combine.

Overall, we find that VAST meets our single-machine performance expections. In particular, we prioritized ab-

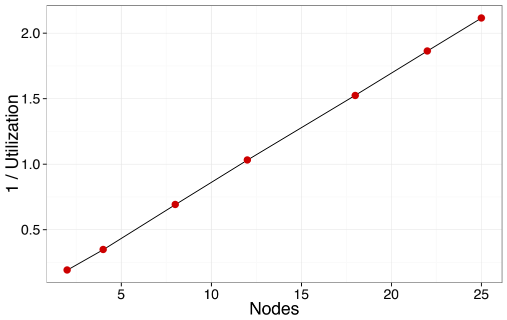


Figure 11: Per-node CPU utilization during ingestion.

straction to performance in our implementation, and have not micro-optimized code bottlenecks (such as via inspecting profiler call graphs). Given that each layer of abstraction—from low-level bit-wise operations to high-level concurrency constructs—comes at the cost of performance, we believe that future tuning efforts hold promise for even further gains.

5.3 Scaling

In addition to single-machine benchmarks, we analyze how VAST scales over multiple machines in a cluster setting, as this will constitute the only viable deployment model for large sites exhibiting copious amounts of data.

Ingestion. Our first measurement concerns quantifying how CPU load during event import varies as a function of cluster nodes. To this end, we ingest 1.24 B Bro connection logs by load-balance them over the cluster NODEs in batches of 65 K. That is, as in Figure 1(b), a SOURCE on a separate machine parses the logs and generates batches with a median rate of 125 K events per second. Due to the fixed input rate, we assess scaling by looking at the CPU load of each worker.

Figure 11 shows per-machine CPU inverse utilization $1/U$ for $U = \sum_i^N (u_i + s_i) / \sum_i^N t_i$ with user CPU time u_i, system CPU time s_i, and wallclock time t_i, for selected values of i in $[0, N]$. The value U can exceed 1.0 because each node runs several threads, and CPU time measurements yield the sum of all threads. As one would expect for effective load-balancing, we observe linear scaling gains for each added node N.

Query. Our second measurement seeks to understand how query latency changes when varying the number of nodes. We show the index completion time of query D in Figure 12. For these measurements, we first primed the file system cache in each case to compensate for a shortcut that our current implementation takes (it maintains the index in numerous small files that cause high seek penalties for reads from disk; an effect we could avoid by optimizing the disk layout through an intermediary step so that the index can read its data sequentially).

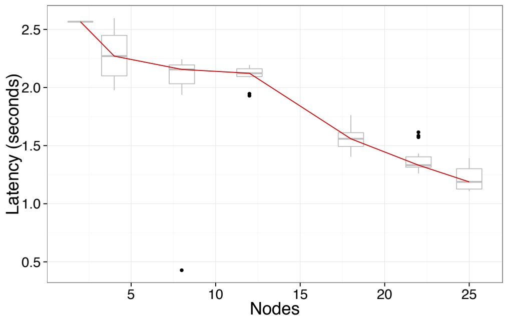


Figure 12: Index completion latency as function of nodes.

We observe linear scaling from 12 nodes upward, but experience problems for the lower half. Other queries show linear scaling for small numbers of nodes. We are in the process of investigating the discrepancy.

5.4 Storage

Unlike systems which process data in situ, VAST relies on secondary indexes that require additional storage space. In the case of the Bro connection logs, the index increases the total storage by 90%. VAST, however, also compresses the raw data before storing in the archive, in this case cutting it down to 47% of its original size. Taken together, VAST requires 1.37 times the volume of its raw input. For PCAP traces VAST, archives entire packets, but skips all packet payload during index construction. Archive compression brings down the trace to 92% of its original size, whereas the index for connection 4-tuple plus timestamps amounts to 4%. In total, VAST still occupies less space than the original data.

String and container indexes require the most storage, due to their composite and variable-length nature. The remaining indexes exhibit constant space design, and their concrete size is a direct function of encoding and layout of the bit vectors.

6 Conclusion

When security analysts today attempt to reconstruct the sequence of events leading to a cyber incident, they struggle to bring together enormous volumes of heterogeneous data. We present VAST [54], a novel platform for forensic analysis that captures and retains a high-fidelity archive of a network's *entire* activity, leveraging domain-specific semantics to manage high data volumes while supporting rapid queries against historical data. VAST's novelty comes from synthesizing powerful indexing technology with a distributed, entirely asynchronous system architecture that can fully exploit today's highly concurrent architectures. Our evaluation with real-world log and packet data demonstrates the system's potential to support interactive investigation and exploration at a level beyond what current systems offer.

References

[1] AGARWAL, R., KHANDELWAL, A., AND STOICA, I. Succinct: Enabling Queries on Compressed Data. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2015).

[2] ALLMAN, M., KREIBICH, C., PAXSON, V., SOMMER, R., AND WEAVER, N. Principles for Developing Comprehensive Network Visibility. In *Proceedings of the USENIX Workshop on Hot Topics in Security (HotSec)* (2008).

[3] ANTOSHENKOV, G. Byte-aligned Bitmap Compression. In *Proceedings of the Conference on Data Compression (DCC)* (1995), p. 476.

[4] ARMSTRONG, J. *Making Reliable Distributed Systems in the Presence of Software Errors*. Ph.D. Thesis, Department of Microelectronics and Information Technology, KTH, Sweden, 2003.

[5] BAO, C., HUITEMA, C., BAGNULO, M., BOUCADAIR, M., AND LI, X. IPv6 Addressing of IPv4/IPv6 Translators. RFC 6052, Internet Engineering Task Force (IETF), 2010.

[6] BAYER, R., AND MCCREIGHT, E. M. Organization and Maintenance of Large Ordered Indexes. In *Record of the ACM SIGFIDET Workshop on Data Description and Access* (1970), pp. 107–141.

[7] CHAMBI, S., LEMIRE, D., KASER, O., AND GODIN, R. Better Bitmap Performance with Roaring Bitmaps. *CoRR abs/1402.6407* (2014).

[8] CHAN, C.-Y., AND IOANNIDIS, Y. E. Bitmap Index Design and Evaluation. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1998), pp. 355–366.

[9] CHAN, C.-Y., AND IOANNIDIS, Y. E. An Efficient Bitmap Encoding Scheme for Selection Queries. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (1999), pp. 215–226.

[10] CHAROUSSET, D., SCHMIDT, T. C., HIESGEN, R., AND WÄHLISCH, M. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proceedings of the International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!)* (2013).

[11] COHEN, M. I., BILBY, D., AND CARONNI, G. Distributed Forensics and Incident Response in the Enterprise. *Digital Investigations 8* (2011), S101–S110.

[12] COLANTONIO, A., AND DI PIETRO, R. CONCISE: Compressed 'n' Composable Integer Set. *Information Processing Letters 110*, 16 (2010), 644–650.

[13] CORRALES, F., CHIU, D., AND SAWIN, J. Variable Length Compression for Bitmap Indices. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA)* (2011), pp. 381–395.

[14] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the Conference on Symposium on Operating Systems Design & Implementation (OSDI)* (2004), vol. 6, pp. 10–10.

[15] DELIÈGE, F., AND PEDERSEN, T. B. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2010), pp. 228–239.

[16] DOHERTY, W. J., AND THADANI, A. J. The Economic Value of Rapid Response Time. *IBM* (1982).

[17] ElasticSearch. `https://www.elastic.co/products/elasticsearch`.

[18] etcd. `https://github.com/coreos/etcd`.

[19] FUSCO, F., DIMITROPOULOS, X., VLACHOS, M., AND DERI, L. pcapIndex: An Index for Network Packet Traces with Legacy Compatibility. *SIGCOMM Computer Communication Review 42*, 1 (2012), 47–53.

[20] FUSCO, F., STOECKLIN, M. P., AND VLACHOS, M. NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1382–1393.

[21] FUSCO, F., VLACHOS, M., AND DIMITROPOULOS, X. RasterZip: Compressing Network Monitoring Data with Support for Partial Decompression. In *Proceedings of the Internet Measurement Conference (IMC)* (2012), pp. 51–64.

[22] GIURA, P., AND MEMON, N. NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)* (2010), pp. 277–296.

[23] GUZUN, G., AND CANAHUATE, G. Performance Evaluation of Word-Aligned Compression Methods for Bitmap Indices. *Knowledge and Information Systems* (2015), 1–28.

[24] GUZUN, G., CANAHUATE, G., CHIU, D., AND SAWIN, J. A Tunable Compression Framework for Bitmap Indices. In *In Proceedings of the International Conference on Data Engineering (ICDE)* (2014), pp. 484–495.

[25] HEWITT, C., BISHOP, P., AND STEIGER, R. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)* (1973), pp. 235–245.

[26] HOARE, C. A. R. Communicating Sequential Processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[27] INGO MLLER, CORNELIUS RATSCH, F. F. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the International Conference on Extending Database Technology (EDBT)* (2014), pp. 283–294.

[28] ISO/IEC. Information technology – Microprocessor Systems – Floating-Point arithmetic. Standard 60559:2011, 2011.

[29] Apache Kafka. `http://kafka.apache.org`.

[30] KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching*, vol. 3. 1998.

[31] KREIBICH, C., WEAVER, N., NECHAEV, B., AND PAXSON, V. Netalyzr: Illuminating the Edge Network. In *Proceedings of the Internet Measurement Conference (IMC)* (2010), pp. 246–259.

[32] LEACH, P. J., MEALLING, M., AND SALZ, R. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, Internet Engineering Task Force (IETF), 2005.

[33] LEE, J., LEE, S., LEE, J., YI, Y., AND PARK, K. FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2015).

[34] LEMIRE, D., KASER, O., AND AOUICHE, K. Sorting Improves Word-aligned Bitmap Indexes. *Data & Knowledge Engineering 69*, 1 (2010), 3–28.

[35] libpcap. `http://www.tcpdump.org`.

[36] Lucene. `https://lucene.apache.org`.

[37] LZ4: Extremely Fast Compression algorithm. `https://github.com/Cyan4973/lz4`.

[38] MAIER, G., SOMMER, R., DREGER, H., FELDMANN, A., PAXSON, V., AND SCHNEIDER, F. Enriching Network Security Analysis with Time Travel. In *Proceedings of the ACM SIGCOMM conference* (2008).

[39] MARTNEZ-PRIETO, M. A., BRISABOA, N., CNOVAS, R., CLAUDE, F., AND NAVARRO, G. Practical Compressed String Dictionaries. *Information Systems 56* (2016), 73–108.

[40] MELNIK, S., GUBAREV, A., LONG, J. J., ROMER, G., SHIVAKUMAR, S., TOLTON, M., AND VASSILAKIS, T. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 330–339.

[41] O'NEIL, P. E. Model 204 Architecture and Performance. In *Proceedings of the International Workshop on High Performance Transaction Systems* (1987), pp. 40–59.

[42] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2014), pp. 305–319.

[43] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks 31*, 23–24 (1999), 2435–2463.

[44] ROTEM, D., STOCKINGER, K., AND WU, K. Optimizing Candidate Check Costs for Bitmap Indices. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)* (2005), pp. 648–655.

[45] SCHMIDT, A., AND BEINE, M. A Concept for a Compression Scheme of Medium-Sparse Bitmaps. In *Proceedings of the International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)* (2011), pp. 192–195.

[46] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST)* (2010), pp. 1–10.

[47] SINHA, R. R., AND WINSLETT, M. Multiresolution Bitmap Indexes for Scientific Data. *ACM Transactions on Database Systems (TODS) 32*, 3 (2007).

[48] Snappy: A fast compressor/decompressor. `https://code.google.com/p/snappy/`.

[49] Splunk. `http://www.splunk.com`.

[50] STOCKINGER, K., CIESLEWICZ, J., WU, K., ROTEM, D., AND SHOSHANI, A. Using bitmap index for joint queries on structured and text data. *New Trends in Data Warehousing and Data Analysis* (2009), 1–23.

[51] TAYLOR, T., COULL, S. E., MONROSE, F., AND MCHUGH, J. Toward Efficient Querying of Compressed Network Payloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (2012), USENIX ATC '12.

[52] VALLENTIN, M. *Scalable Network Forensics*. Ph.D. Thesis, University of California, Berkeley, 2016. (*in preparation*).

[53] VAN SCHAIK, S. J., AND DE MOOR, O. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *Proceedings of the International Conference on Management of Data (SIGMOD)* (2011), pp. 913–924.

[54] VAST. `http://vast.io`.

[55] WONG, H. K. T., LIU, H.-F., OLKEN, F., ROTEM, D., AND WONG, L. Bit Transposed Files. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (1985), pp. 448–457.

[56] WU, K., OTOO, E., AND SHOSHANI, A. On the Performance of Bitmap Indices for High Cardinality Attributes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)* (2004), pp. 24–35.

[57] WU, K., OTOO, E. J., AND SHOSHANI, A. A Performance Comparison of Bitmap Indexes. In *Proceedings of International Conference on Information and Knowledge Management (CIKM)* (2001), pp. 559–561.

[58] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2012), pp. 2–2.

Appendix

In §4.2.2 we introduce the structure of VAST's high-level index types, but do note elaborating how to operate on them with concrete algorithms. Table 3 and Table 4 provide this information in more detail. For each type, we show how to append a value (symbol $\twoheadleftarrow$) to an index as well as how to query it in terms of logical operations.

The *basic* index I in these tables represents a bitmap or inverted index with a fixed binning, encoding, and compression scheme. I operates on unsigned integer values and supports operators $\{<, \leq, =, \neq, \geq, >\}$. For append and lookup algorithms of the concrete encoding schemes, we refer the reader to the literature [8, 9], from which summarize established results about multi-component indexes in the following.

The basic index forms the foundation for the k-component index $K^\beta = \langle I_1, \dots, I_k \rangle$ with $|\beta| = k$ (see §4.2.1). It represents the foundation of many higher-level indexes and its lookup algorithm varies according to the relational operator of the predicate. Answering equality queries involves computing a simple conjunction:

$$EQ(i,x) = \bigwedge_{j=1}^{i} (I_j = x_j) \tag{1}$$

Thus, we can answer $K^\beta = x$ with $EQ(k,x)$, and $K^\beta \neq x$ with $\overline{EQ(k,x)}$. For one-sided range queries, the algorithm *less-than-or-equal* (LE) implements range lookup of the form $K^\beta \leq x$ as follows:

$$LE(i,x) = \begin{cases} (I_i \leq x_i - 1) \vee (\theta_i \wedge LE(i-1,x)) & i > 1, x_i > 0 \\ \theta_i \wedge LE(i-1,x) & i > 1, x_i = 0 \\ (I_i \leq x_i - 1) \vee LE(i-1,x) & i > 1, x_i = \beta_i - 1 \\ I_i \leq x_i & i = 1 \end{cases} \tag{2}$$

The extra parameter θ_i depends on the coding scheme and means either $I_i = x_i$ or $I_i \leq x_i$. Putting together algorithms EQ and LE, we can now answer $K^\beta \circ x$ under all relational operators with the algorithm ℓ:

$$\ell(K^\beta, \circ, x) = \begin{cases} EQ(k,x) & \circ \in \{=\} \\ \overline{EQ(k,x)} & \circ \in \{\neq\} \\ LE(k,x) & \circ \in \{\leq\} \\ \overline{LE(k,x)} & \circ \in \{>\} \\ LE(k,x-1) & \circ \in \{<\} \wedge x > 0 \\ \overline{LE(k,x-1)} & \circ \in \{\geq\} \wedge x > 0 \\ \mathbb{0} & \circ \in \{<\} \wedge x = 0 \\ \mathbb{1} & \circ \in \{\geq\} \wedge x = 0 \end{cases} \tag{3}$$

The two results $\mathbb{0}$ and $\mathbb{1}$ denote the empty and complete identifier set. In the case of bitmap indexes, $\mathbb{0}$ represents a bit vector with all 0-bits, whereas $\mathbb{1}$ only consists of 1-bits.

Type	Structure	Append	Lookup
basic*	I	$I \twoheadleftarrow x^{(\alpha)}$	$I \circ x$
k-component†	$K^\beta = \langle I_k, \ldots, I_1 \rangle$ $\Theta^k = K^\beta \quad \beta_i{=}\beta_j{=}2 \wedge \lvert\beta\rvert{=}k$ $\Phi^w = K^\beta \quad \Pi_{i=1}^k \beta_i \leq 2^w$	$K^\beta \twoheadleftarrow x^{(\alpha)} \equiv I_i \twoheadleftarrow x_i^{(\alpha)} \quad \forall 1 \leq i \leq k$	$K^\beta \circ x \equiv \ell(K^\beta, \circ, x)$
`bool`	$\mathbb{B} = S$	$\mathbb{B} \twoheadleftarrow x^{(\alpha)} \equiv S \twoheadleftarrow \alpha \quad$ iff $x = \texttt{true}$	$\mathbb{B} \circ x \equiv \begin{cases} \overline{S} & x = \texttt{false} \\ S & x = \texttt{true} \end{cases}$
`count`	$\mathbb{C} = \Phi^{64}$	$\mathbb{C} \twoheadleftarrow x^{(\alpha)} \equiv \Phi^{64} \twoheadleftarrow x^{(\alpha)}$	$\mathbb{C} \circ x \equiv \Phi^{64} \circ x$
`int`	$\mathbb{I} = \Phi^{64}$	$\mathbb{I} \twoheadleftarrow x^{(\alpha)} \equiv \Phi^{64} \twoheadleftarrow (x^{(\alpha)} \overset{u}{+} 2^{63})$	$\mathbb{I} \circ x \equiv \Phi^{64} \circ (x \overset{u}{+} 2^{63})$
`real`§	$\mathbb{F} = \langle S, E, M \rangle$ $S = \mathbb{B}$ $E = \Theta^{11}$ $M = \Theta^{52}$	$\mathbb{F} \twoheadleftarrow \langle x_s, x_e, x_m \rangle^{(\alpha)} \equiv \begin{cases} S \twoheadleftarrow x_s^{(\alpha)} \\ E \twoheadleftarrow x_e^{(\alpha)} \\ M \twoheadleftarrow x_m^{(\alpha)} \end{cases}$	$\mathbb{F} \circ x \equiv \begin{cases} S = x_s \wedge E \circ x_e \wedge M \circ x_m & \circ \in \{=, \neq\} \\ S = 0 \wedge E \circ x_e \wedge M \circ x_m & x \geq 0 \wedge \circ \in \{>, \geq\} \\ S = 1 \vee (E \circ x_e \wedge M \circ x_m) & x \geq 0 \wedge \circ \in \{<, \leq\} \\ S = 0 \vee (E \triangledown x_e \wedge M \triangledown x_m) & x < 0 \wedge \circ \in \{>, \geq\} \\ S = 1 \wedge E \triangledown x_e \wedge M \triangledown x_m & x < 0 \wedge \circ \in \{<, \leq\} \end{cases}$
`duration`	$\mathbb{D} = \mathbb{I}$	$\mathbb{D} \twoheadleftarrow x^{(\alpha)} \equiv \mathbb{I} \twoheadleftarrow x^{(\alpha)}$	$\mathbb{D} \circ x \equiv \mathbb{I} \circ x$
`time`	$\mathbb{T} = \mathbb{D}$	$\mathbb{T} \twoheadleftarrow x^{(\alpha)} \equiv \mathbb{D} \twoheadleftarrow x^{(\alpha)}$	$\mathbb{T} \circ x \equiv \mathbb{D} \circ x$
`string`	$\mathbb{S} = \langle \phi, \kappa_1, \ldots, \kappa_M \rangle$ $\phi = K^\beta$ $\kappa_i = \Theta^8$	$\mathbb{S} \twoheadleftarrow \langle x_1, \ldots, x_n \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \kappa_i \twoheadleftarrow x_i^{(\alpha)} \end{cases} \forall 1 \leq i \leq n$	$\mathbb{S} \circ x \equiv \begin{cases} o & \lvert x\rvert > M \\ \phi = 0 & \lvert x\rvert = 0 \\ \phi = \lvert x\rvert \wedge \bigwedge_{i=1}^{\lvert x\rvert} \kappa_i = x_i & \circ \in \{=, \neq\} \\ \phi \geq \lvert x\rvert \wedge \bigvee_{i=1}^{M-\lvert x\rvert+1} \left(\bigwedge_{j=1}^{\lvert x\rvert} \kappa_{i+j-1} = x_j \right) & \circ \in \{\in, \notin\} \end{cases}$

[*] The *basic* index has a fixed binning, coding, and compression scheme and operates on values $x \in X \subseteq \mathbb{N}_0^+$ and has cardinality $C \leq |X|$. (see §4.2.1)

[†] The k-component index operates with a base $\beta = \langle \beta_k, \ldots, \beta_1 \rangle$. We show algorithm ℓ in §**??**. The bit-sliced index [55] is a special of K^β where $\beta_i = \beta_i = 2$ for all $i \neq j$. The multi-component index Φ^w can at most represent 2^w distinct values.

[§] We denote by $\triangledown$ the "mirrored" operator of $\circ$, e.g., $<$ and $>$.

Table 3: Summary of append and lookup operations on high-level indexes.

Type	Structure	Append	Lookup
`addr`	$\mathbb{A} = \Theta^{128}$	$\mathbb{A} \twoheadleftarrow \langle x_1, \ldots, x_{128} \rangle^{(\alpha)} \equiv \Theta_i^{128} \leftarrow x_i^{(\alpha)} \quad \forall 1 \le i \le 128$	$\mathbb{A} \circ x \equiv \begin{cases} \bigwedge_{i=1}^{k} \Theta_i^{128} = x_i & \circ \in \{\in\} \\ \overline{\mathbb{A} \ni x} & \circ \in \{\notin\} \end{cases}$
`subnet`	$\mathbb{U} = \langle \mathbb{A}, \Phi^8 \rangle$	$\mathbb{U} \twoheadleftarrow \langle x_a, x_p \rangle^{(\alpha)} \equiv \begin{cases} \mathbb{A} \twoheadleftarrow x_a^{(\alpha)} \\ \Phi^8 \twoheadleftarrow x_p^{(\alpha)} \end{cases}$	$\mathbb{U} \circ x \equiv \begin{cases} \Phi^8 \le x_p \wedge \left(\bigwedge_{i=1}^{p} \mathbb{A}_i = x_{a_i} \right) & \circ \in \{\in\} \\ \overline{\mathbb{U} \ni x} & \circ \in \{\notin\} \end{cases}$
`port`	$\mathbb{P} = \langle \Phi^{16}, \Phi^2 \rangle$	$\mathbb{P} \twoheadleftarrow \langle x_n, x_t \rangle^{(\alpha)} \equiv \begin{cases} \Phi^{16} \twoheadleftarrow x_n^{(\alpha)} \\ \Phi^2 \twoheadleftarrow x_t^{(\alpha)} \end{cases}$	$\mathbb{P} \circ x \equiv \begin{cases} \Phi^{16} \circ x_n \wedge \Phi^2 = x_t & x_t \neq \texttt{unknown} \\ \Phi^{16} \circ x_n & x_t = \texttt{unknown} \end{cases}$
`vector`[*]	$\mathbb{X}^V = \langle \phi, \mathbb{V}_1, \ldots, \mathbb{V}_M \rangle$ $\phi = K^\beta$	$\mathbb{X}^V \twoheadleftarrow \langle x_1, \ldots, x_n \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \mathbb{V}_i \twoheadleftarrow x_i^{(\alpha)} & \forall 1 \le i \le n \end{cases}$	$\mathbb{X}^V \circ x \equiv \begin{cases} \text{see } \mathbb{S} \circ x & \tau(x) = \texttt{vector} \\ \text{see } \mathbb{X}^S \circ x & \tau(x) = \texttt{set} \end{cases}$
`set`	$\mathbb{X}^S = \langle \phi, \mathbb{V}_1, \ldots, \mathbb{V}_M \rangle$ $\phi = K^\beta$	$\mathbb{X}^S \twoheadleftarrow \langle x_1, \ldots, x_n \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \mathbb{V}_i \twoheadleftarrow x_i^{(\alpha)} & \forall 1 \le i \le n \end{cases}$	$\mathbb{X}^S \circ x \equiv \begin{cases} \mathrm{o} & \lvert x \rvert > M \\ \phi = 0 & \lvert x \rvert = 0 \\ \bigwedge_{i=1}^{\lvert x \rvert} \left(\bigvee_{j=1}^{M} \mathbb{V}_j = x_i \right) & \mathit{otherwise} \end{cases}$
`table`[†]	$\mathbb{X}^T = \langle \phi, X_1, \ldots, X_M, Y_1, \ldots, Y_M \rangle$ $\phi = K^\beta$ $X_i = \mathbb{V}$ $Y_i = \mathbb{V}$	$\mathbb{X}^T \twoheadleftarrow \langle (k_1, v_1) \ldots, (k_n, v_n) \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ X_i \twoheadleftarrow k_i^{(\alpha)} & \forall 1 \le i \le n \\ Y_i \twoheadleftarrow v_i^{(\alpha)} & \forall 1 \le i \le n \end{cases}$	$\mathbb{X}^T \circ k \equiv \begin{cases} \bigvee_{i=1}^{M} X_i = k & \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni k} & \circ \in \{\notin\} \end{cases}$ $\mathbb{X}^T \circ v \equiv \begin{cases} \bigvee_{i=1}^{M} Y_i = v & \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni v} & \circ \in \{\notin\} \end{cases}$ $\mathbb{X}^T \circ (k, v) \equiv \begin{cases} \bigvee_{i=1}^{M} (X_i = k \wedge Y_i = v) & \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni (k \to v)} & \circ \in \{\notin\} \end{cases}$

[*] Depending on the type $\tau(x)$ of value x, the lookup function can either preserve ordering (as in substring search) or ignore ordering (as in subset search).

[†] A table value has the form $x = \langle (k_1, v_1), \ldots, (k_n, v_n) \rangle$. We show lookups for a single key, value, or mapping.

Table 4: Summary of append and lookup operations on high-level indexes.