Scalable Network Forensics

by

Matthias Vallentin

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Vern Paxson, Chair
Professor Michael Franklin
Professor David Brillinger

Spring 2016

Scalable Network Forensics

Abstract

Scalable Network Forensics

by

Matthias Vallentin

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Vern Paxson, Chair

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, which often leaves 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 descriptions of past activity recorded over extended time. A typical analysis starts with a narrow piece of intelligence, such as 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 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 that investigations often involve. While standard Security Information and Event Management (SIEM) solutions 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.

In this thesis 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 offers a type-rich data model to avoid loss of critical semantics, allowing operators to express activity directly. Similarly, strong typing persists throughout the entire system, enabling type-specific optimization at lower levels while retaining type safety during querying for a less error-prone interaction.

A central contribution of this work concerns our novel type-specific indexes that directly support the type's common operations, e.g., top-k prefix search for IP addresses. We show that composition of these indexes allows for a powerful and unified approach to fine-grained data localization, which directly supports the workflows of security investigators. 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. Our evaluation with real-world log and packet data demonstrates the system's potential to support interactive exploration at a level beyond what current systems offer. We release VAST as free open-source software under a permissive license.

To my parents

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgments

I would like to express my sincere gratitude to a number of people who made this work possible. Their continuous support, feedback, and encouragement provided the nourishing environment for this dissertation to take on shape.

My heartfelt appreciation goes to Vern Paxson. His deeply respectful and tactful treatment of his students made having him as my advisor an outstanding experience. I feel incredibly fortunate to have had countless opportunities to interact with him. His visionary ideas, coupled with a unique sense for abstraction and analytical rigor, never cease to inspire me.

Additionally, for over a decade, I am delighted to have known Robin Sommer as a colleague, mentor, and friend. He patiently listened to all my fledgling architectural ideas and helped separate the wheat from the chaff. His extensive experience and invaluable advice had a profound impact on this project.

My gratitude extends to Seth Hall, whose unstoppable enthusiasm and desire to reap the fruits from this work fueled my motivation and grounded it in an operational context. I am also indebted to Dominik Charousset and his work on the C++ Actor Framework. Our numerous technical discussions forged an enjoyable, organic collaboration.

Several others played a major role in completing this work. My full and genuine thanks to David Brillinger and Michael Franklin for serving on my dissertation committee. Their advice and comments significantly strengthened this work.

I have received great support from the International Computer Science Institute (ICSI). In particular, I am grateful for my fabulous two office mates Johanna Amann and Mobin Javed, who always had an open ear for my unformed ideas. I thank the whole ICSI staff for making this such a pleasant stay, and in particular Maria Eugenia Quintana for her healthy, positive attitude. Also thanks to Christian Kreibich for our brainstorming sessions on devising apt plots for visualizing data, and to Nicholas Weaver for reliably playing devil's advocate.

Moreover, my thanks go to the operational security team at the Lawrence Berkeley National Laboratory (LBNL), and particularly Aashish Sharma for his real-world perspectives on incident response. Special thanks to Yahel Ben-David for his support on everything wireless, to Keith Lehigh for sharing insights from his immensely large network, to Gregory Bell for his stimulating ideas, and to Mark Allman for coining the term VAST. Many others impacted this work. In particular, I thank Samir Al-Sheikh, Justin Azoff, Scott Campbell, Vlad Grigorescu, Andreas Reuter, Fabrice Ryba, Robert Schmidt, Thomas Schmidt, Audrey Sillers, Pedro Simoes, Adam Slagell, Vincent Stoffer, and Matthias Wählisch. This work was supported by National Science Foundation grants 0716640, 1161799, 1237265, 1348077, and by U.S. ARL MURI grant W911NF-09-1-0553.

Finally, I express my deepest gratitude and admiration to my parents and my sister. Without their unconditional love and perpetuating support, I would have never reached this point.

Chapter 1

Introduction

One cannot not communicate.
Paul Watzlawick

Large networks never sleep. Computers continuously emit a stream of activity as an artifact of their communication. On the one hand, humans generate this activity, fueled by their deep desire for omnipresent connectivity. Mobile devices aggressively attempt to jump on available wireless networks, retrieving their latest content feeds while sending upstream a wealth of collected user telemetry in order to deliver an even richer, more personalized, and fully captivating user experience. To keep up with these steep user demands, network operators of enterprise networks continually expand their backend infrastructure by providing more computational resources, storage capacity, and bandwidth. On the other hand, closed feedback control systems operate independently of human input, relying on a wealth of sensors to steer decision making. Microwaves, cars, and power plants receive more and more networking capabilities to automatically monitor and synchronize state changes. Overall, today's countless devices form an omnipresent communication network, perpetually relaying information.

The network attack surface increases as a side effect of this endless growth. Operators struggle to defend both a network's perimeter and its deep interior. The larger network, the more complex it behaves and the harder it becomes to defend. Once connected to the Internet, any device unwillingly exposes itself to a horde of attackers waiting for the user to make a mistake: by visiting a malicious website, leaving open a vulnerable service, or simply ignoring security indicators which warn about potentially intercepted communication. Attacks constitute a constituent part of large networks. Operators and security staff spend a significant amount of time quarantining infected hosts, notifying administrators, and implementing protective measures to prevent future occurrences of similar attacks.

In addition to the continuous process of establishing secure network operations, security analysts also perform retrospective analysis of attacks. During the investigation of a security incident, the network activity footprint becomes essential data. Analysts rely on it, 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 [8].

This process, however, suffers from *fragmentation across space*: analysis remains manual and time-consuming, as no single system efficiently integrates the numerous sources of data (e.g., system status messages, firewall alerts, notices from intrusion detection systems, network monitoring logs) that investigations often involve. The lack of uniform analysis procedures forces security analysts to agglomerate relevant activity from disparate sources to construct a coherent picture, with cumbersome and error-prone ad-hoc data processing methodologies. Existing security information and event management (SIEM) solutions aggregate logs from different sources into a single database, but their data models omit crucial type semantics, and they struggle to scale to the data rates that large-scale environments require.

Furthermore, this process suffers from *fragmentation across time*: existing methods of processing events relating to past activity starkly differ from how analysts express activity occurring in the future. For example, sifting through logs to find an infected machine follows a very different methodology compared to configuring a firewall to block the same incident in the future. This discrepancy prevents analysts from "closing the loop" efficiently: after a successful post-mortem investigation, an analyst must transfer the gained insight to a domain with different idiosyncrasies.

Based on these needs, and drawing upon extensive experience working closely with operational security staff, we set out in this thesis to address the inherent deficiencies of this process: we design and implement VAST (Visibility Across Space and Time), a system to streamline network forensics at scale.

1.1 Use Cases

The term *forensic* stems from the Latin word *forensis*, the adjective corresponding to the noun *forum*. In Roman times, the forum was a marketplace to exchange goods, as well as a social place to give speeches and exchange opinions [100]. During criminal trials, the forum served as court of justice, hearing and deciding disputes [79, 167]. Therefore, the adjective forensic has predominantly a juridical connotation today.

More broadly, *forensics* (or forensic science) as a discipline encompasses methods and means to collect evidence during an investigation, often in a criminal context. *Digital forensics* applies this notion to digital technology and crime scenes [159]. This thesis has *network forensics* as its central theme, a branch of digital forensics which involves reconstructing the activities which led to a security incident [185].

Our perspectives regarding the practice and requirements for network forensics stem from close contact with operational security staff at the Lawrence Berkeley National Laboratory (LBL) and the Bro [147, 30] community, which includes numerous incident responders and operators from large networks. In the process we have assisted and observed forensic analyses and gained a deeper understanding of prevalent use cases in the domain, which we introduce in the following.

1.1.1 Incident Response

After a security breach occurs, the tasked analyst must quickly isolate the scope and impact of the issue to provide actionable determinations, such as the need to remove connectivity, rebuild systems, and/or reauthenticate users. The analysis often starts with a narrow piece of intelligence, typically a local system exhibiting questionable or clearly problematic behavior, or a report from another site (facing similar threats) of a particular sequence of activity that they identified as associated with a successful attack on their own systems. Sometimes the information is less discriminating, such as a new list of malicious IP addresses, domains, URLs, or malware executables.

Starting with this intelligence, the analyst tries to locate the described behavior by examining logs of past activity. Often this entails cross-correlating entries from multiple logs of different types. Upon locating the activity directly corresponding to the intelligence, the analyst then begins interactive procedure of gathering additional context associated with the activity—again drawing upon multiple logs—and inspecting the findings for relevance. Not infrequently this context in turn suggests additional exploration to undertake, for example by producing new items of intelligence relating to how an attacker behaved or how to locate additional systems exhibiting similar activity. At this point the analysis *iterates* (“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.

To close the loop after completing the investigation of an incident, the analyst ideally codifies the accumulated knowledge (for example, the full set of contextual indicators of a system compromise or the presence of the attacker) and installs rules in the site's security monitoring to receive notifications if the activity ever occurs in the future. Doing so may further involve *what-if* checks on historic log data to gauge the degree to which these rules may result in false positives.

Today, the process of configuring the monitor works fully separate from the analysis procedure; the analyst must hand-translate the findings from their investigation in order to express their detection in the rule language(s) used by the site's monitoring.

1.1.2 Network Troubleshooting

Configuration bugs rarely manifest in an obvious manner, rather, operators have to pinpoint the error by sifting through heaps of logs from diverse sources. The more context and perspectives operators can draw upon, the more confidence in the analysis develops, but the layered, modular structure of network architecture and protocols renders this process a complex task.

When being confronted with a high-level user complaint (consider the example of email system failures), the operator typically kicks off the analysis by spot-checking a dashboard with aggregate statistics about the present state. These aggregates cover information across the whole network stack, ranging from link-layer to application-specific details, and ideally exist in various temporal granularities, such as seconds, hours, and days. A natural representation of this data is in the form of time series, allowing operators to apply numerous statistical techniques for data exploration and correlation. For example, an operator may look at aggregates of mail traffic to find a significant drop in connections, and then check DNS requests involving the mail server to notice that the DNS server generates only sporadic replies. Finally, the operator inspects the DNS activity more closely, revealing that the server process entered an out-of-memory crash cycle due to a torrent of requests from infected machines in a student dormitory.

Unlike incident response, which exhibits a *bottom-up* characteristic in that typical analysis begins with a concrete fact and then turns into a wider search, network troubleshooting resembles a *top-down* process, starting from abstract symptoms requiring deeper investigation.

1.1.3 Insider Abuse

Insider abuse is difficult to apprehend because *authorized* actions form a policy violation only when analyzed in context. To illustrate, consider an employee accessing a sensitive document on an internal machine, copying it to their laptop, and then exfiltrating it via email from there. While each action in isolation does not appear to reflect malicious activity, the sequence of actions constitutes a policy violation. Investigating potential instances of such attacks often require reinspecting activity previously deemed benign, again using a "peeling the onion" approach.

A carefully orchestrated insider attack often manifests over long periods of time, with piecemeal extraction of sensitive data to stay under the radar of monitoring systems. From a detection standpoint, this requires relating specific, temporally distant data points. The analyst must stitch together pieces of evidence to reconstruct puzzle pieces into a coherent chain of actions. Looking for descriptions of activity that substantiate the case frequently appears like searching for a needle in a haystack.

The Snowden leaks have demonstrated the efficacy of a carefully orchestrated exfiltration process. Today's massive storage systems can archive copious amounts of data, but it remains an open challenge to efficiently access and correlate the subsets relevant to an investigation.

1.2 Goals

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

1.2.1 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. Per §1.1, forensic investigations exhibit a highly iterative workflow, in which analysts repeatedly query a system to inspect various data subsets to support their case. Often, a "taste" of the full query result suffices to perform a decision in this triaging phase. We thus postulate query "taste" latencies in the order of seconds for the analysis to remain viable.

Analyst time is a scarce and costly resource. The more cases security staff can solve per unit time, the higher the return on invest. Aside from the economic benefit, an interactive system also boosts productivity [66]—a synergy which amplifies the utility of forensic analysts.

1.2.2 Scalability

Large networks generate a torrent of data relevant for post-facto investigations. Archiving and retrieving this data requires a scalable system, which efficiently utilizes available resources—both within a single machine as well as across multiple machines.

For intra-machine scalability, a system should carefully multiplex its computational tasks over the available CPU cores, as well as schedule I/O accesses to intelligently harness persistent storage devices without incurring high latencies. For inter-machine scalability, a system should distribute its data and computation over multiple machines. Adding a new node to a system should increase its capacity linearly.

But scaling does not only imply growing in size and power, it also means shrinking when the system runs over-provisioned. For example, a system which replicates its components over multiple machines for redundancy or load-balancing could consolidate its footprint by migrating state to a smaller set of machines, allowing for powering off the freed capacities.

1.2.3 Expressiveness

Data comes in many formats and some exhibit more structure than others. For example, one event may represent simply the unavailability of a resource as a boolean flag, while another contains detailed, hierarchical information about protocol data (e.g., a HTTP request with query parameters belonging to a TCP session represented as connection 4-tuple). Since network forensics encompasses correlation of data from various sources, a supporting system must accommodate and represent them uniformly without losing structural information.

In addition to retaining structure, keeping and assigning type information during data import avoids losing domain-specific semantics, which can prove valuable later at search time. Rich typing also allows analysts to work within their domain, as opposed to spending time translating their workflows to lower-level system primitives. Moreover, a strict type system enables powerful system optimizations. For example, when an analyst looks for an IP address instead of a plain string, type-specific optimizations can speed up the search.

1.3 Outline

We structure this thesis according to Figure 1.1 and briefly summarize the remaining chapters in the following.

§2: Background. We begin with summarizing the status quo of network forensics, finding a strikingly thin treatment of this field in an academic context. In §2.2, we then branch out to related work in the systems community, to assemble the building blocks necessary to build a scalable system for network forensics. In particular, we look at a high-performance messaging substrate in §2.3 and indexing techniques to accelerate search in §2.4.

§3: Architecture. In this chapter we present the design of our VAST system which enables scalable network forensics. After introducing VAST's rich-typed data model in §3.1, we present in §3.2 each component in detail.

§4: Implementation. In this chapter we look under the hood VAST and highlight key aspects of the implementation. We discuss challenges with the underlying message-passing framework in §4.1 and then shift our attention to high-level indexing in §4.2. Thereafter, we explain in §4.3 how VAST achieves an efficient query processing engine.

§5: Evaluation. In this chapter we evaluate VAST along several dimensions. After introducing our measurement infrastructure and data sets in §5.1, we show how we ensure correctness of operation in §5.2. In §5.3, we examine throughput during data ingestion and indexing. In §5.4, we look at the responsiveness of queries by examining the execution pipeline. We seek to understand intra-machine and inter-machine scaling in §5.5, and finish with a quantification of storage overhead in §5.6.

§6: Conclusion. The last chapter concludes the thesis. We summarize key insights and lessons learned in §6.1. Throughout the course of this dissertation project numerous fruitful ideas beyond the scope of this thesis emerged, which we sketch in §6.2.

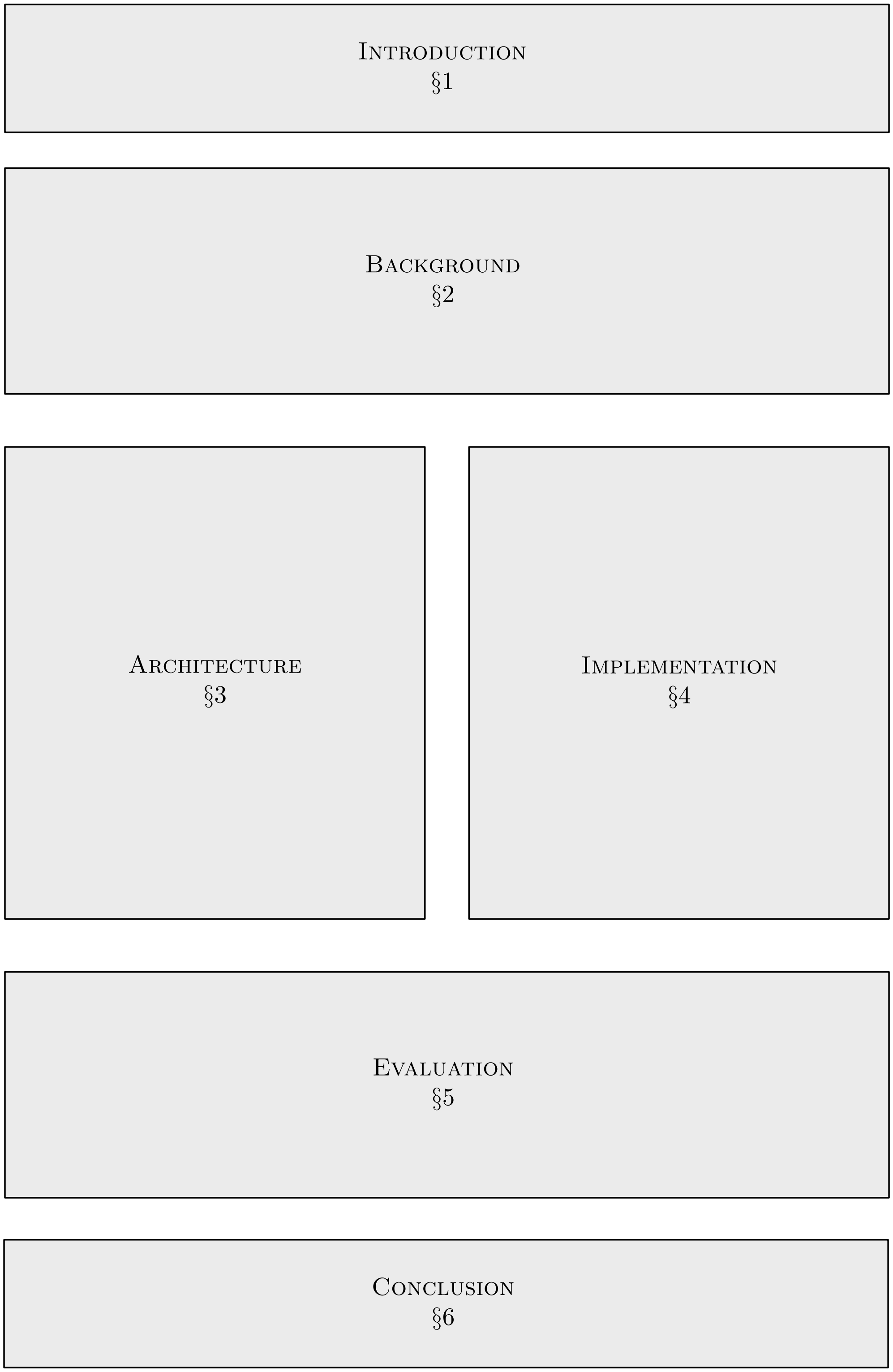


Figure 1.1: Thesis structure.

Chapter 2

Background

Technological progress has merely provided us
with more efficient means for going backwards.
Aldous Huxley

Before designing and implementing a system for network forensics, we take a close look at existing solutions, analyzing their contributions and shortcomings. In §2.1, we begin with a comprehensive study of the field of network forensics from an academic standpoint. Afterwards, we broaden our view in §2.2 to discuss existing technology for data-intensive applications at scale. We summarize work ranging from traditional databases, modern data stores, and frameworks for distributed computation. Then we step down to a lower level of system design and study the communication layer of distributed systems. In §2.3 we focus on flexible message passing abstractions that form the cornerstone of our system, enabling it to scale both within a single machine as well as over a cluster of multiple machines. Efficient search is another pillar in a system which must perform lookups in massive data archives. In §2.4 we summarize apt database technology to answer high-dimensional queries.

2.1 Literature Search

The academic treatment of large-scale network forensics is strikingly thin. We concluded this after comprehensively examining 4,795 papers and 181 presentations of 13 security and systems conferences on forensics (FIRST, DFRWS), security (USENIX Security, ACM CCS, IEEE S&P, NDSS, ACSAC, USENIX HotSec), databases (VLDB), and systems (USENIX LISA, USENIX ATC, SOSP, USENIX NSDI). As we illustrate in Figure 2.1, the last decade of research on network forensics paints a fragmented picture: only occasional interest, even in security-centric venues. Let us walk through the existing landscape.

The Digital Forensic Research Workshop (DFRWS) annual conference primarily covers traditional host-based forensics, with network forensic only appearing occasionally. For example,

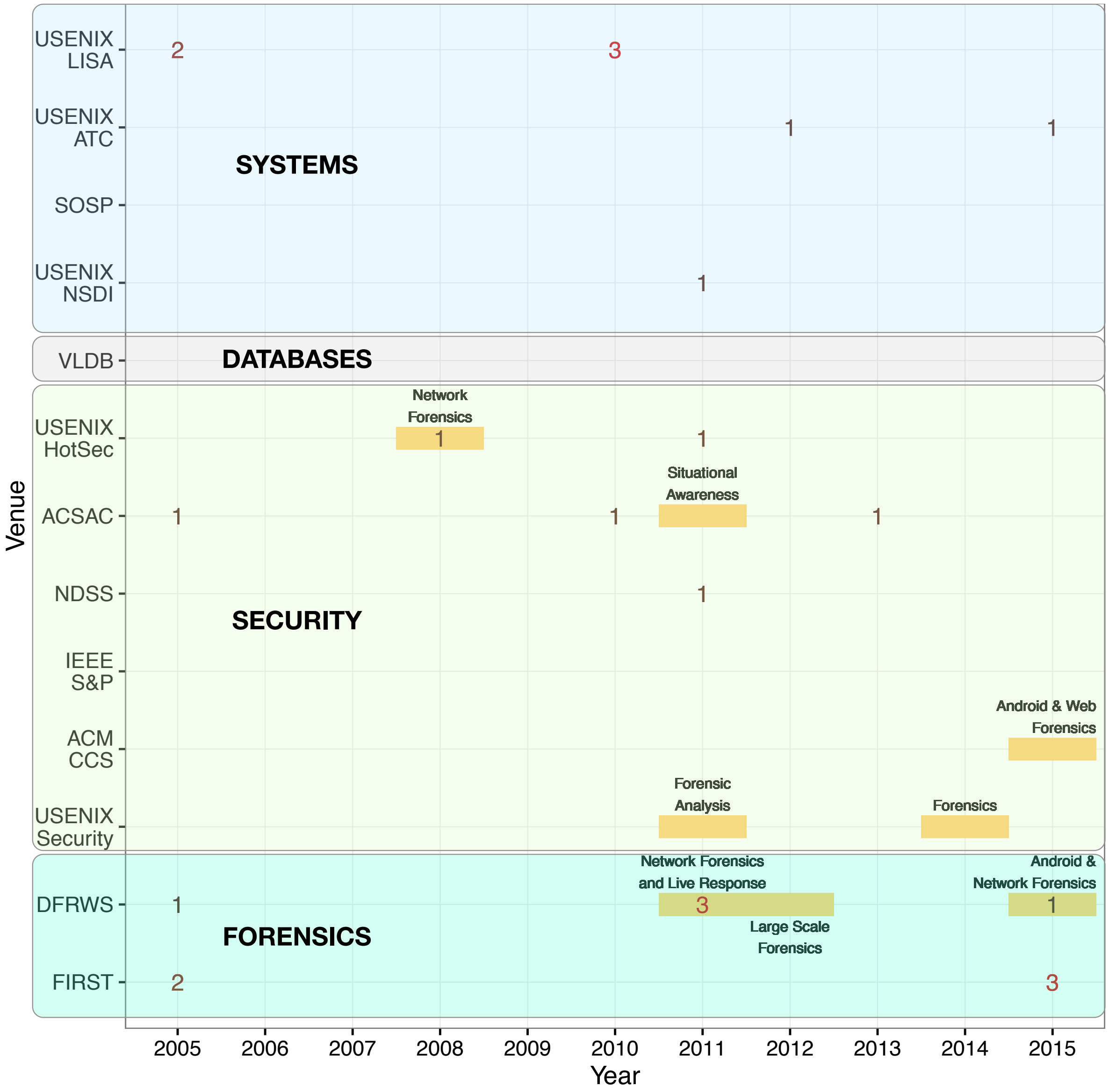


Figure 2.1: A summary of research on network forensics over the last decade in relevant venues, based on analysis of 4,795 papers and 181 presentations. Each number reflects the publications per venue relevant to central network forensics issues, which sum up to 13 in total.

the session Network Forensics and Live Response in 2011 covered file carving [22], anonymizing NetFlow data [169], and presented a framework for network-wide "live" forensics [51], i.e., investigating an incident by looking at state from running machines, as opposed to searching through archives of past data. In the following year, the session Large Scale Forensics emphasized the scaling aspect, but merely at the scope of host-level forensics [83, 68]. The session Android & Network Forensics in 2015 introduced work on mobile-device network forensics at small scales [194]. Similarly, the Forum of Incident Response and Security Teams (FIRST) hosts annual meetings where field exports give presentations. FIRST covered unified logging in 2005 [171], but the very theme of network forensics did not come up until 2015, and then with a narrow, small-scale focus [98], or high-level experience report from a proprietary system [101, 27]. Even forensics-centric venues like DFRWS and FIRST do not appear to face the underlying systems challenges that arise with network forensics at larger scales.

In 2011 and 2014, USENIX Security featured a Forensics Analysis session, but with a focus on single-machine forensics. The Annual Computer Security Applications Conference (ACSAC) in 2005 included work on network forensics, which utilizes evidence graphs for multi-stage attack detection [196]. In 2010, ACSAC covered host-based forensics in one paper, and in 2011, a Situational Awareness session—without an emphasis on network forensics. In 2013, one paper presented techniques to automatically normalize and extract knowledge from enterprise logs (e.g., IP-to-host mappings and communication profiles), as well as support the incident response process by flagging suspicious users based on behavioral analysis [206]. The USENIX HotSec workshop in 2008 featured a Network Forensics track where a concept paper [8], written by colleagues, precisely laments the insufficiently available tooling and proposes guidelines for designing an apt system. In fact, this paper marks the starting point for this thesis project. In 2011, one HotSec paper emphasizes the research opportunities in digital forensics, because existing solutions does not translate into readily usable technology [193], or simply do not scale [82]. In 2012, the USENIX Annual Technical Conference (ATC) published work on a key issue in network forensics [181]: the absence of adequate tools to support the domain-specific workflows efficiently (see §2.2.4). In 2015, USENIX ATC brought back the theme, although with a narrower focus on network-level flows [119]. At the USENIX Large Installation System Administration (LISA) conference in 2005, the operationally driven need for a unified representation of authentication log data came up [168], as well as the topic of network awareness, albeit at the level of network packets only [102]. Field practitioners regularly articulate the need for a unified view on activity [49, 171, 82, 191], and we explicitly state this aspect as a necessary design goal for a network forensics system in §1.2. In 2010, LISA brought back network awareness along with a packet-oriented tool for passive service discovery and heuristics-based intrusion detection [21], as well as the theme of unified log analysis [153] and correlation [115]. While these topics relate to important aspects in network forensics, the presented solutions do not support the iterative peeling-the-onion workflow of security analysts. The 2011 USENIX Symposium on Networked Systems Design and Implementation (NSDI) featured research which introduces a high-level framework for network event analysis [191]. The proposed language allows for expressing composable behavior models, which represent a sequence of states. In addition to traditional boolean filter operations, the language supports notions of causal behavior relationships, partial/total ordering, concurrent operations, and value dependencies. Implementing such a framework into a scalable system for interactive forensic analyses remains an open challenge for future work.

Overall, our examination of existing work in the field suggests that network forensics received very selective attention. The 13 venues we inspected include a total of 13 publications relating to central network forensics questions. We witness several attempts to improve the status quo in this domain, but available solutions fail to meet *systems challenges* we postulate in §1.2: network forensics at scale requires both real-time data ingestion as well as an interactive query engine grounded in a flexible data model. Therefore, we now take take a closer look at available technology at the database and systems community, with the goal of identifying architectural building blocks to assemble a system that can meet the requirements of network forensics at scale.

In §2.2, we conduct a survey of related work, ranging from databases, over modern distributed computing frameworks, as well as more research specific to network forensics outside the previously examined venues. Thereafter, we turn to two architectural cornerstones we deem crucial for network forensics at scale: high-performance message passing (§2.3) and indexing to accelerate search (§2.4).

2.2 Related Work

In this section we contextualize related work, evaluating existing solutions with respect to their aptitude for the domain per the uses cases in §1.1. Since network forensics involves handling massive amounts of data, the following discussion concentrates on work in databases (§2.2.1 and §2.2.2), distributed systems (§2.2.3), as well as domain-specific solutions (§2.2.4). We summarize the main aspects of the existing landscape with respect to the requirements of network forensics in Table 2.1.

2.2.1 Traditional Databases

Traditional relational database management systems (DBMS) exist since the 1970s and still find wide application today in scenarios requiring transactional semantics with strong consistency guarantees. These systems use the popular structured query language (SQL) as universal means to access data, which expresses queries in a flexible, declarative style [37, 106]. Two common DBMS workload distinctions have emerged over time: online transactional processing (OLTP) and online analytical processing (OLAP) systems. OLTP workloads consist of numerous short transactions in the form of inserts, updates, and deletes. System evaluations typically express the performance of OLTP systems in transactions per second. OLAP workloads occur in data warehouses [46] and consist of complex, high-dimensional queries over read-mostly data sets. Data imports often take place periodically in batches, following the extract-transorm-load (ETL) paradigm [189]. A common database deployment not only keeps day-to-day operations data (such as employees, salaries, clients, orders) in a

		Capabilities								
		Generic				Network Forensics				
		Generic Computation	Stream Processing	Complex Queries	Horizontal Scalability	Continuous Ingestion	Rich Data Model	Type Safety	Interactive Search	Iterative Analysis
Technologies	Relational DBMS (e.g., PostgreSQL [150], MySQL [138], Oracle [144])	✗	✗	✓	✗	✗	✓	✓	✓	✗
	Streaming Databases (e.g., Borealis [1], STREAM [11], TimeStream [152])	✗	✓	✓	✗	✓	✓	✓	✓	✗
	Data Warehouses (e.g., Hive [183], Greenplum [50], Avatara [203])	✗	✗	✓	✓	✗	✓	✓	✓	✓
	NoSQL Stores (e.g., Redis [156], MongoDB [137], Riak [160])	✗	✗	✗	✓	✓	✗	✗	✗	✗
	MapReduce (e.g., Hadoop [94])	✓	✗	✓	✓	●*	✓	✓	✗	✗
	Distributed Computation (e.g., Spark [209], DryadLINQ [208])	✓	✓	✓	✓	✓	✓	✓	●†	●†
	Log Aggregators (e.g., splunk [24])	✗	✗	✗	✓	✓	✓	✗	✓	✗
	VAST	✗	✗	✗	✓	✓	✓	✓	✓	✓

[*] Decoupled from query execution engine.
[†] Only when working set fits into memory.

Table 2.1: Related work evaluated with respect to its aptitude for the domain of network forensics.

relational DBMS which faces OLTP workloads, but also for decision support and analytics a data warehouse, which exhibits OLAP workloads.

While DBMS execute *historical queries* over data that already exists on permanent storage, data stream management systems (DSMS) [35, 87] answer *continuous queries* that operate on data after issuing the query. Put differently, a historical query transports the query to the data, whereas a continuous query routes the data through the query. Once the original data has traveled past a continuous query, the engine discards it. Thus, streaming databases do not offer persistence, besides snapshotting query state and meta data. DSMS often rely on synopsis data structures (e.g., HyperLogLog [74, 96], Count-Min Sketch [56], Bloom filters [26, 179]) to succinctly summarize and retain certain aspects of the original data.

Per §1.1, in the domain of network forensics analysts gather evidence to arrive at a best possible explanation for an incident. This reasoning involves investigation of immutable descriptions of activity from the past. Modification of data in the OLTP sense does not occur. (However, modification may occur during aging out old data in more space-efficient representations.) Therefore, data warehousing and OLAP-style workloads predominate in network forensics. However, conventional ETL systems only support bulk loading, as opposed to continuously integrating data as it arrives. When it comes to archiving network activity, there exists no opportunity to interrupt the query execution and bulk-load the system. Queries must execute along in the presence of a continuous ingestion process.

Streaming databases have also relevance for network forensics. While post-facto analysis examines past data via historical queries, incident response further encompasses a preventative aspect, namely to ensure that the investigated security incident or similar attack can be detected again in the future. To close the loop, the analyst installs the developed historical query as continuous query, ideally by simply flipping a switch. In this sense, the domain requires a streaming capability in addition to the data warehousing capability. The idea to unify the two analysis styles is not new [158], but existing prototypes do not scale beyond a single machine and do not offer interactive, OLAP-style querying.

2.2.2 Modern Data Stores

Modern data-hungry applications demand high scalability and performance, which traditional DBMS cannot provide due to their monolithic system architecture. NoSQL stores [36] have emerged as a new breed of databases to specifically overcome these limitations by focusing on (*i*) horizontal scalability, (*ii*) a simple programmatic interface, typically in the form of key-value access, (*iii*) weaker consistency than ACID (atomicity, consistency, isolation, durability), commonly referred to as BASE (basically available, soft state, eventually consistent), and (*iv*) replication and fault tolerance via partitioning and replication. NoSQL stores primarily find application in OLTP workload scenarios [177], but typical network forensics queries exhibit OLAP characteristics: users ask fine-grained high-dimensional queries as opposed to performing sheer key-value lookups. Consequently, we only consider NoSQL stores as a building block as part of a bigger system, instead of solution by itself; e.g., in conjunction with indexing where a lookup yields keys to access the base data.

Unlike NoSQL stores, data warehouses [46] aim for answering high-dimensional queries over large bodies of immutable data in a timely manner. Leading vendors in this space, such as Oracle, Teradata, SAP [163], and EMC Greenplum [50], address the growing demand for large-scale analytics in real-time with proprietary parallelization techniques. Big Internet companies, such as Google, Facebook, Twitter, LinkedIn, assemble their big data stack out of a combination of techniques to cover a broad range of use cases [166]. Their ecosystem features dedicated frameworks for cluster management, data storage, and distributed processing—including specific solutions for data warehousing [183, 184, 118, 203, 95].

A particularly notable data warehouse is Google's Dremel [133], a system which stores semi-structured data in situ in a columnar format, using a SQL interface for ad-hoc queries having interactive response times. Queries execute in a serving tree structure where intermediate nodes aggregate results from their children. This hierarchical form of aggregation enables an efficient scalable execution platform outside the MapReduce [59] paradigm.

Another data warehouse from Google is Mesa [91], which supports real-time data import and query functionality, designed for fault-tolerant, multi-datacenter deployments. The system exhibits a relational data model, supports aggregation at various resolutions, and can handle evolving schemata. Like many large-scale systems from Google, Mesa leverages existing infrastructure: Colossus, the successor of the Google File System (GFS) [85], as distributed filesystem for bulk storage, BigTable [42] for storing meta data, and MapReduce [59] for query execution.

VAST shares similarities with the objectives of Dremel and Mesa: answering high-dimensional queries consisting of multiple predicates over large bodies of immutable data. While Mesa stores its data in tables and relies on indexes primarily to accelerate seek times, VAST exclusively uses indexes to compute query results. This improves latency, but comes at the cost of a limited query language imposed by the index operations.

Vertica offers a commercial version [116] of their column store C-Store [176]. Like Mesa, the system aims for supporting both analytic real-time workloads and transactional updates. The open-source data warehouse Druid [205] also relies on columnar storage. To accelerate filter lookups, Druid additionally uses indexes. The system architecture places specific functionality into dedicated nodes responsible for, e.g., data ingestion, historical queries, and client interaction. Unlike Druid, VAST separates deployment from program logic, which allows for flexible operation in a single operating system process or over a cluster of machines. VAST also relies on indexes to accelerate search, but does not decompose the base data into columnar format. Instead VAST stores the base data in compressed blocks in a key-value store. Although this design provides only row-based access and not arbitrary query execution, it simplifies sharing of the base data with other applications. In the future, we may investigate decomposition of the base data in columnar format or interface it with other storage layers.

ElasticSearch [69] is a commercial, document-oriented database built on top of Apache Lucene [126], which provides a full-text inverted index (see §2.4.2). ElasticSearch effectively wraps Lucene behind a RESTful [73] API. Partitioning and replication provide horizontal scaling and fault-tolerance. Since all data arrives in JSON format, ElasticSearch uses "mappings" to define a schema that translates JSON to Lucene data types. Similarly, Solr [173] builds on top of Lucene to provide a distributed indexing solution. As of this writing, Lucene (and thereby ElasticSearch and Solr) does not support IPv6 addresses, which renders it unsuitable for network forensics today. It neither supports schema evolution, nor handles container types. VAST features a flexible, semi-structured data model and has support for IPv6 addresses as well as containers. Moreover, VAST can ingest high-volume streams of input data, ranging from raw network packets to high-level application logs.

2.2.3 Distributed Computing

Aside from database technology, distributed compute architectures provide scalable general-purpose execution platforms to efficiently process large bodies of data. These systems offer data-parallel execution engines spanning multiple nodes. These architectures focus less on sophisticated storage structures, but rather on distributed algorithms.

The MapReduce [59] paradigm remains the cornerstone of distributed compute frameworks. Popularized through its open source implementation Hadoop [198], the model enables computation of arbitrary functions over a cluster of commodity machines using two primitives: a `map` function to generate intermediate key-value pairs, and a `reduce` function to merge pairs sharing the same key. The runtime orchestrates the distributed execution of these functions, handles data transfers and inter-node communication, and guarantees fault-tolerance by restarting failed jobs on different machines. MapReduce jobs execute off a distributed filesystem [85, 170, 146] in which the runtime also stores the result. This incurs high I/O load during the initial full data scan and when storing the result at the end. Although one can avoid touching the filesystem for intermediate results by relying on in-memory pipelines [55], the fundamentally batch-oriented nature of this execution model remains. In network forensics, a typical workflow involves several iterations over a query until having located the relevant amount of information. With MapReduce-based execution, each query iteration translates into a separate job, which cannot deliver an interactive experience.

The availability of Hadoop spawned numerous high-level query engines [148, 139, 183, 23] which offer declarative languages which compile queries into a sequence of MapReduce jobs, and therefore inherit the same limitations. Dryad [105] provides a domain-specific language consisting of higher-level constructs to build an acyclic dataflow graph. In contrast to MapReduce where the runtime sets up the dataflow graph, Dryad requires users to specify the low-level communication patterns. The framework also targets batch processing and materializes results to the filesystem.

Spark [209] distributes data over the main memory of available machines in a cluster. After having pre-loaded a working set in memory, the provided functional primitives allow for efficient data sharing between stages of computation. Spark introduces the notion of a resilient distributed dataset (RDD), a parallel data structure which provides fault-tolerance by tracking the transformations applied to it. In addition to operating on preloaded working sets, Spark also offers a streaming mode in which computations occur in small deterministic batches on top of the RDD model [210]. Spark also offers a Succinct RDD [4], an optimization for search, which stores the base data in situ in compressed, flat files. This representation does not require decompression when searched. Succinct leverages suffix trees internally, which support 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 inappropriate for high-volume scenarios such as network forensics, where a continuous stream of data arrives. When the working set fits in memory, Succinct offers competitive performance, but not when primarily executing off the filesystem.

In summary, security analysts can rarely define a working set *a priori*, which can result in thrashing due to frequent loading and evicting data from memory. For example, assume that data partitions reside on the filesystem sorted by time. Since there can only be one sort order, a lookup along a different dimension (e.g., a spatial aspect, such as an IP address) can exhibit poor locality and spread over numerous partitions. A complex query may even require a full scan of all partitions, in which case distributed in-memory computing provides no benefit over MapReduce execution. Therefore, we envision VAST going hand-in-hand with Spark, where VAST quickly pinpoints a tractable working set and then hands it off to Spark for more complex analysis if needed.

2.2.4 Network Forensics Domain

In addition to general-purpose systems, there exist several approaches which aim for directly supporting the inherent workflows of the network forensics domain. Most approaches operate on raw network traffic and offer search capabilities at varying granularity.

The Bro network security monitor [147, 30] offers a high-level platform for network analysis along with a type-rich, event-based scripting language. Bro reassembles raw packets into transport-layer byte streams, which protocol analyzers then dissect into fine-grained streams of application-specific events. User can write handlers for these events to perform arbitrary computation. Bro also ships with a library of scripts which record the protocol activity in detailed log files. In previous work, we developed the Bro cluster [187] to scale the analysis to multi-Gbps links. In this mode, a load-balancer dispatches the stream of packets over a set of backend nodes, such that packets from the same connection arrive at the same node. Since Bro produces only log files and does not come with a persistence component, manual search quickly runs into scalability issues. VAST complements Bro by providing a scalable solution for persistence that can natively ingest its logs.

The Time Machine [130] records raw network traffic and builds tree indexes (see §2.4.1) for a limited set of packet headers fields. To cope with large traffic volumes, the system tracks connections and foregoes packets belonging to the same flow after the connection byte stream has reached a certain threshold. The choice of tree-based indexes prevents efficient composition of hits. For example, querying both IP source and destination address requires an index which spans both fields. Such a design does not scale to higher dimensions. Similarly, pcapIndex [75] indexes packet headers, but relies on bitmap indexes (see §2.4.2) for better composability. VAST represents a superset of both Time Machine and pcapIndex: it supports the same cutoff functionality. In VAST, packets simply constitute a special type of input.

FloSIS [119] provides a higher-level interface to bulk traffic storage at the granularity of flows instead of packets. To efficiently access flow data, FloSIS uses a two-stage indexing approach. At the first level, 4 Bloom filters [26] (for the connection 4-tuple) and 2 timestamps (for beginning and end) track whether a data partition does not contain the queried data and can be skipped. At the second level, sorted arrays of flow meta data provide a logarithmic lookup method to the base data. Similar to the Time Machine and pcapIndex, the system operates exclusively on network traffic and exhibits an architecture only suitable for this form of data.

NetStore [86] is a column store also geared towards flow archiving. It includes two indexes to accelerate search: (*i*) an interval tree to select the appropriate partition based on a temporal constraint, and (*ii*) inverted indexes over the connection 5-tuple. NetStore can handle insertion rates on the order of 10 K records/second, and exhibits average query latencies on the order of 10s of seconds over a data set containing 62 M records, each of which contains 12 fixed-size numeric fields. These performance figures remain too low for an interactive query experience at already moderate data volumes, and the system architecture does not scale beyond single-machine deployments.

NET-Fli [76] is a single-machine NetFlow indexer which leverages bitmap indexes to accelerate search. The systems comes with a promising (though patented) bit vector encoding scheme, COMPAX. NET-Fli sustains import rates up to 0.5–1.2 M NetFlow records per second (with 12 numeric fields per record), with room for an order-of-magnitude improvement when offloading index construction to a GPU [77]. From a high-level view, VAST exhibits a similar conceptual architecture: the base data resides in an archive, with horizontally partitioned bitmap indexes pointing back into it. However, VAST differs in several salient points. First, VAST integrates this dichotomous archive-index design into a distributed architecture to scale beyond single machine deployments. Second, rather than targeting only specific data sources, such as packets or NetFlow records, VAST offers a generic data model which then maps to type-specific bitmap index layouts. Third, we distribute VAST as open-source software under a permissive BSD licence free of patents, allowing the community to reuse it as a building block for higher-level applications.

The separation of base data into archive and partitioned index also exists in other systems [181]. A notable variation includes a separate meta index to identify the relevant main index partitions to load during a query. This multi-stage indexing resembles multi-resolution indexes [172], except that the top-level index points to partitions as opposed to individual data records.

Splunk [24] is a commercial log aggregation system focusing on time-series data. Unlike many other systems, splunk applies typing at search time while internally keeping data as plain strings. As new event data arrives, splunk tokenizes it into keywords and builds inverted indexes over the keywords as well as over event meta data [136]. For a query execution backend, splunk implements its own MapReduce engine which operates on partitions divided by time. The late binding of type information incurs significant performance hits during result materialization, especially when dealing with massive data volumes. The lack of internal typing also prevents type-specific storage and query optimizations. Furthermore, splunk cannot dynamically adapt its use of CPU resources to changes in workload, e.g., users statically configure the numbers of threads per indexers. VAST follows the opposite approach strong typing enables type-specific optimizations and provides a type-safe query language to prevent analysts from making subtle mistakes.

In summary, we find that existing work in the field consists of carefully engineered low-level solutions to solve the problem for a specific type of input. However, comprehensive network forensics draws from numerous data sources. We believe a system must not tailor its architecture to a specific data format, and instead should operate on higher-level notions of data types. Some systems exhibit a promising architecture that separates base data and composable indexes. For a scalable system, we must integrate these ideas into a distributed system. In the following, we shift the focus away from holistic systems and discuss the lower-level primitives required to design a scalable distributed system.

2.3 High-Level Message Passing

Network forensics at scale requires processing massive volumes of data in short time spans. To handle the high data ingestion rates while supporting interactive search, a system in this domain must meet ambitious throughput and latency requirements—by resorting to either expensive, special-purpose hardware or a cluster of commodity machines. The latter model often qualifies as the only choice for operators in face of limited budgets.

When designing a distributed system, performance and scale become chief concerns. How well does the system utilize the native parallelism of modern multi-core CPUs? Does it scale up linearly with the number of nodes, and does it also scale *down* to smaller workloads? By scaling down, we mean low fixed overhead and effortless deployment in small setups. A flexible, efficient architecture exploits all available resources, both within the same machine as well as across multiple instances.

Today, developers rely on vastly different techniques to bridge the gap between intra-machine and inter-machine scaling. On the one hand, performance-attentive developers employ low-level concurrency primitives (e.g., tasks, threads, semaphores) to squeeze out maximum performance on a single machine. The implementation results in complex, tightly coupled, and difficult to compose systems. On the other hand, deploying software across machines involves network communication in the form of message passing. Socket management and serialization account for a substantial fraction of the code base, with complicated platform-specific event-loop APIs that result in *inversion of control*: a fragmentation of program logic due to callback registration with the asynchronous runtime.

Combining these two styles in a single application creates a complex code base, difficult to maintain and communicate to new contributors. Developers should not have to rely on system-specific idiosyncrasies to harness the available resources of modern hardware and networks. Instead, a flexible runtime should offer a unified abstraction that allows for expressing concurrent control flow safely, independent of deployment, and without sacrificing performance. In §2.3.1 we present the architectural primitive to achieve this goal, and describe how we select an implementation optimal for our purposes in §2.3.2.

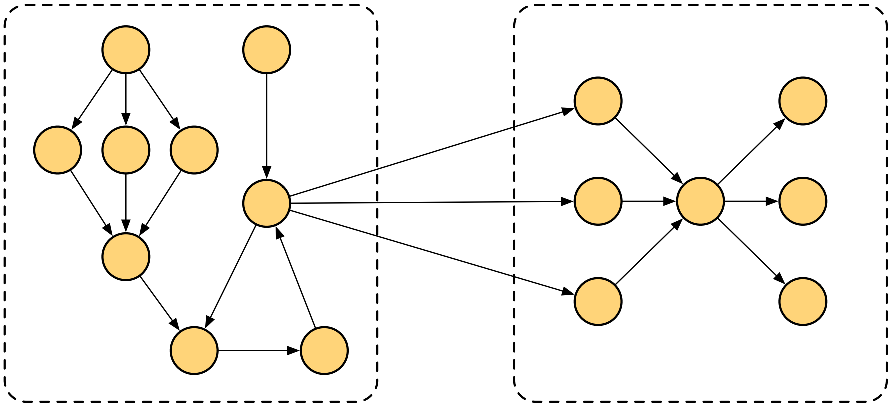


Figure 2.2: The actor model [97]. Each vertex represents an actor. All actors execute concurrently, but each instance sequentially processes one message at a time. The dotted frames represent process boundaries. Actors can both communicate within the same process as well as talk to actors in different processes or remote nodes.

2.3.1 Actor Model

The *actor model* [97] offers a paradigm that unifies concurrency and distribution. In this model, computational entities—called *actors*—execute independently and in parallel. Using unique, location-independent addresses, actors communicate asynchronously solely via message passing. Because actors do not share state, data races cannot occur by design. Each actor posseses a message queue called a *mailbox* [5], from which it dequeues and sequentially processes one message at a time. The actor's *behavior* determines how to process the next message. In response to a message, the behavior can include any of the following three actions: (*i*) create (or *spawn*) new actors, (*ii*) send messages to other actors, or (*iii*) designate the behavior to use for the next message in the mailbox.

Figure 2.2 visualizes an exemplary topology in the form of a directed graph. A vertex represents an actor instance and an edge between two actors indicates that one actor references another, either by spawning or sending messages to it. Throughout this thesis, we reference specific actor types in SMALL CAPS font.

A related model of computation is *communicating sequential processes* (CSP) [99], in which *processes* communicate with other processes via *synchronous channels*. As a result, the sender cannot transmit before the receiver is ready. This creates a tighter coupling compared to the asynchronous fire-and-forget semantics of actor messaging. Moreover, CSP emphasizes the channel as opposed to 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: unlike CSP, the actor model contains a flexible failure propagation model based on monitors, links, and hierarchical supervision trees [13]. 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.

2.3.2 Implementations

Designing a scalable high-performance architecture for modern distributed systems poses an ambitious set of challenges. The actor model provides an excellent fit for this task, but an implementation of this paradigm must meet the following requirements in order to qualify as viable choice in practice.

Native Performance An efficient runtime not only scales across machines, but also maximizes resource utilization within a machine. Many actor runtimes execute in an abstract machine. Most notably, Erlang [12] provided the first industrial-strength implementation as a dynamically typed language, which still finds wide application today. But statically typed languages also feature implementations, such as the Akka framework [7] in Java or Scala.

Type Safety The set of all valid messages an actor can process constitutes its interface. An actor model implementation can either check the message validity at runtime or at compile time. Runtime type checking facilitates rapid prototyping and enables loosely coupled systems, but requires extensive unit testing to ensure correctness. Conversely, compile-time type checking can guarantee correct operation, but comes at the cost of higher development times and tighter coupling, because the sender must have available interface definitions of the receiver.

Heterogeneous Targets Actors offer a single abstraction to separate program logic from deployment. When communicating within the same process, the runtime typically delivers a message by enqueuing a light-weight pointer in the receiver's mailbox. When communicating across process boundaries, the runtime serializes the message, sends it over the network, and deserializes it on the other end before the receiver handles it. In addition to efficient in-memory and network-level messaging, a runtime may also feature GPU actors in the form of OpenCL [143] kernels. This opens up new opportunities to harness the massive SIMD parallelism offered by modern graphics cards, which proves particularly effective for offloading expensive computation, such as index construction [9, 77].

There exist numerous general-purpose implementations of the actor model, each of which makes different design choices in a trade-off space of safety, usability, and performance. Table 2.2 summarizes our comparison of various existing implementations, which we describe according to the features below.

Features

Implementations	Native Execution	Garbage Collection	Pattern Matching	Copy-On-Write Messaging	Failure Propagation	Dynamic Behaviors	Compile-Time Type Checking	Run-Time Type Checking	Exchangeable Scheduler	Network Actors	GPU Actors	Backend
Erlang [13]	✗	✓	✓	✗	✓	✓	✗	✓	✓	✓	✗	BEAM
Elixir [70]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	BEAM
Akka/Scala [7]	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✗	JVM
SALSA Lite [62]	✗	✓	✗	✗	✗	✗	✓	✓	✗	●†	✗	JVM
Actor Foundry [2]	✗	✓	✓	✗	✗	✓	✗	✓	✗	✓	✗	JVM
Pulsar [151]	✗	✓	✓	✗	✓	✓	✗	✓	✗	✓	✗	JVM
Pony [47]	✓	✓	✓	✗	✗	✗	✓	✓	✗	✓	✗	LLVM
Charm++ [109]	✓	●*	✗	✗	✗	✗	✓	✓	✗	✓	✗	C++
Theron [182]	✓	●*	✗	✗	✗	✓	✗	✓	✗	✓	✗	C++
libprocess [124]	✓	●*	✗	✗	✓	✗	✓	✗	✗	✓	✗	C++
CAF [44]	✓	●*	✓	✓	✓	✓	✓	✓	✓	✓	✓	C++

[*] Via reference counting, as opposed to tracing garbage collection.
[†] Only in SALSA, not SALSA Lite.

Table 2.2: Comparison of popular actor model implementations.

Native Execution. Frameworks which compile down to native instructions can deliver superior performance compared to those running in abstract machines, but at the same time cannot provide as effective fault isolation.

Garbage Collection. Many runtimes which execute in abstract machines run with tracing garbage collection as opposed to reference counting. While this simplifies the programming model, it introduces non-deterministic performance spikes which in practice require excessive tuning to achieve comparable performance.

Pattern Matching. Most functional languages offer pattern matching as a first-class primitive to define functions. The ability to express messages as typed tuples and apply pattern matching to select a specific message handler provides a natural form of multiple dispatch in actor model implementations.

Copy-On-Write Messaging. An actor runtime with copy-on-write messaging offers an environment free of data races, since it avoids by design mutable access to message contents from multiple threads of execution. Moreover, multiple actor instances can work with the same message instance without incurring a copy when only performing read-only accesses. Data-intensive applications especially benefit from this feature.

Failure Propagation. The flexible failure propagation in the actor model (via monitors, links, and hierarchical supervision trees [13]) proves highly valuable in large-scale distribution systems.

Dynamic Behaviors. By definition of the actor model, an actor can change its behavior as a reaction to a message. This feature facilitates, for example, the implementation of concepts such as finite state machines.

Compile-Time Type Checking. Static type checking catches message type incompatibilities already during the compilation phase. For example, a program would not compile if an actor sends a message that a receiver cannot handle. The compiler needs access to the type system of the actor runtime to enforce the type checks.

Run-Time Type Checking. A weaker form of type safety offers run-time type checking, where the actor receiving a message determines whether it can handle it. Many library-based solutions offer this guarantee, as it requires less intricate interaction with the host language's type system.

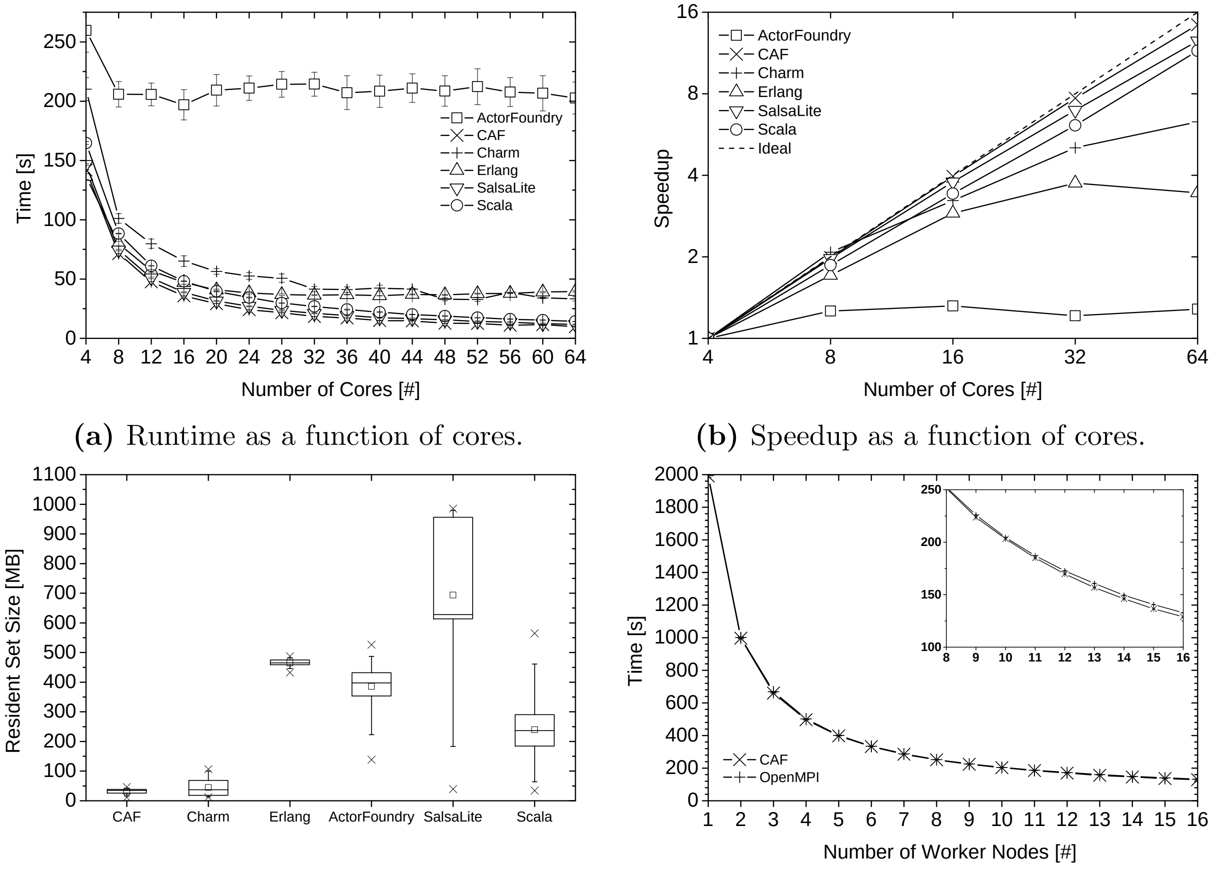
Exchangeable Scheduler. The actor runtime includes a scheduler which maps execution of actors to hardware threads. Some applications require a high throughput while others demand a low latency. A flexible runtime allows for tuning the scheduler according to its needs, e.g., make adjustments regarding throughput, latency, and fairness.

Network Actors. The ability to send a message to an actor on a remote machine in the same way as to a local actor creates a network-transparent system which decouples deployment from logic.

GPU Actors. Spawning actors on general-purpose graphics cards allows for offloading compute-intensive tasks.

Backend. The backend describes the target environment or host language of the actor model implementation.

To better understand the performance characteristics between the different implementations, we summarize experiment results from previous work in Figure 2.3. The C++ Actor Framework (CAF) [44] outperforms all other implementations in terms of CPU performance, memory utilization, and scaling. Even message passing frameworks which operate at lower levels, such as OpenMPI [80], do not exhibit distinguishable performance, as Figure 2.3(d) illustrates. This shows that a high degree of abstraction does not necessarily impose a performance penalty. Moreover, CAF supports both weakly and strongly typed actors, but



(a) Runtime as a function of cores.

(b) Speedup as a function of cores.

(c) Memory consumption during execution.

(d) Overhead compared to OpenMPI [80].

Figure 2.3: CAF performance compared to other actor model implementations [43]. Reproduced with permission. Figures (a)–(c) illustrate a mixed-case scenario, which involves number factorization while stressing the actor runtime at the same time. Figure (d) shows the overhead of CAF compared to OpenMPI [80] for computing Mandelbrot images.

without introducing tight coupling between sender and receiver. To this end, CAF features a globally checked type system where users register message types. CAF also supports subtyping: partial knowledge of an interface suffices to send messages as long as the used subset does not violate the type system. To the best of our knowledge, only CAF supports a general-purpose actor runtime which subsumes in-memory, network-based, and GPU actors under a single, location-transparent programming model. For these reasons, we deem CAF a superior choice for our requirements.

Moreover, we closely collaborated with the developers since the inception of CAF and continue to do so on a regular basis. Indeed, use cases from VAST motivated several features available in CAF today. This close collaboration allowed us not only to harness a powerful message-passing middle layer, but also to shape it according to our needs. CAF also ships with a permissive BSD license.

2.4 Accelerating Search

Forensic analysts often sift through data in an explorative fashion (see §1.1). A selective probe here, following a hunch there. "Why did a workstation connect to the customer database at 3 a.m.? Why does the same machine send a torrent of outbound DNS TXT traffic to a machine across the globe? Oh, apparently this happened every night since last week." The puzzle often reveals itself when putting together all the pieces, where each part can lead to new threads of illuminating evidence.

To perform such analyses, enormous volumes of descriptions of activity must be readily searchable. But sequential scans cannot deliver an interactive query experience at scale, as the data size exceeds the aggregate memory capacity even of clusters of machines. Consequently, supporting the inherently iterative analysis style requires a more selective form of data access.

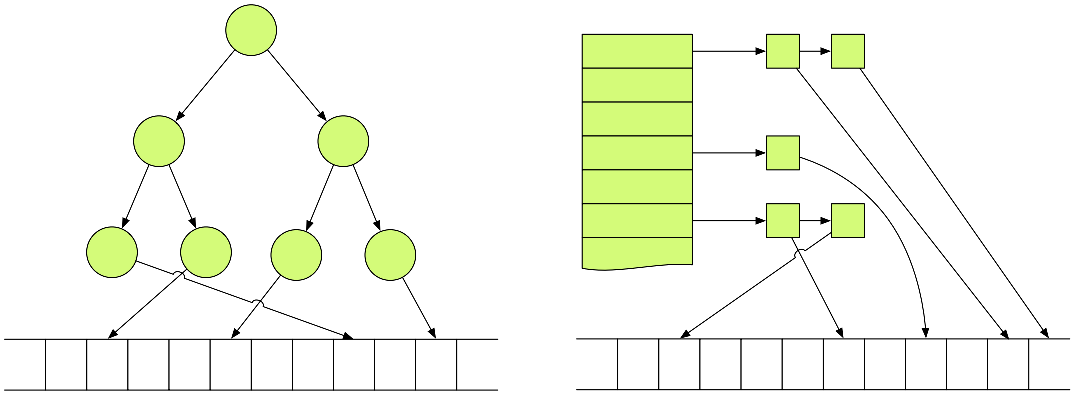
2.4.1 Hash and Tree Indexes

An *index* grants efficient data access through an indirection, at the cost of additional data structure. Common databases indexes,[1] such as B-trees and hash tables [19, 112], provide a fast entry point into the base data via a search key. These indexes constitute the building block of traditional DBMS, due to their well-understood properties and robustness in various workloads. We illustrate their high-level structure in in Figure 2.4. Either a tree or hash table can yield an entry point into the base data at the bottom, from where sequential seeks navigate through the individual records.

However, these traditional indexes do not *compose* efficiently during search in higher dimensions: they require a total order on the key space, which does not exist for more than one dimension. Even special higher-dimensional indexes still suffer from the "curse of dimensionality," i.e., index space usage and lookup time quickly turn into exponential growth as the number of predicates in the query expression increases [20, 81].

In forensic investigations, already simple queries exhibit a high dimensionality. To illustrate, consider the query "give me all outbound DNS requests since last week," which we can model as a boolean expression with three predicates, $A \wedge B \wedge C$, where A represents outbound connections, B DNS requests, and C connections with a timestamp within the last week. From there, additional restrictions of the search space seem plausible, e.g., restrict the search to specific IP addresses, UDP ports, or DNS record types. Thus, even simple analysis quickly spans multiple dimensions.

[1] The database community uses "indexes" more frequently than "indices" as plural of "index." Therefore, we only use "indices" when referring to a set of positions in a sequence, whereas "indexes" when referring to the data structure.



(a) A tree index. **(b)** A hash table index.

Figure 2.4: Efficient access of the base data through an index.

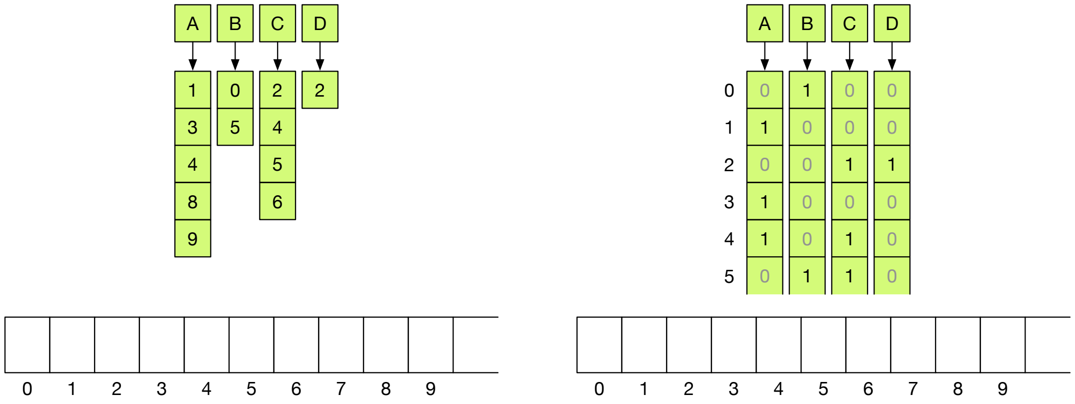
2.4.2 Inverted and Bitmap Indexes

An index that does not suffer from the curse of dimensionality is the *inverted index* [112], the building block of modern information retrieval. The inverted index maps search keys to a set of identifiers representing the unique elements in the base data, as we illustrate in Figure 2.5(a): keys A–D map to lists of identifiers (aka. *position lists*). An isomorphic data structure is the *bitmap index* [199, 141], which we depict in Figure 2.5(b). It also maintains a list of identifiers per lookup key, but in the form of *bit vectors*.[2] The position of a 1-bit represents the identifier. In our simplified example, key B maps to the bit vector $\langle 100001 \rangle$, which corresponds to the set $\{0, 5\}$.

In our inverted index example, answering the query expression $B \vee (C \wedge D)$ translates to retrieving the corresponding position lists $L^B = \{0, 5\}$, $L^C = \{2, 4, 5, 6\}$, $L^D = \{2\}$, and applying set operations on them: $L^B \cup (L^C \cap L^D) = \{0, 2, 5\}$. Assuming the inverted index maps keys to sorted arrays, the set operations involve merging the arrays. Decades of research went into the question of how to perform these operations efficiently [103, 72, 18, 61, 15]. Similarly, the bitmap index evaluates the expression $B \vee (C \wedge D)$ by retrieving the bit vectors $B^B = \langle 100001 \rangle$, $B^C = \langle 0010011 \rangle$, $B^D = \langle 0010011 \rangle$, and then computing bitwise $B^B \mid (B^C \ \& \ B^D)$. As the bit vector size and number of operations increase, it becomes important to choose an efficient evaluation algorithm [175, 201, 121]. Unlike tree and hash indexes, which point directly into the base data, inverted and bitmap index lookups scale linearly with the number of predicates in an expression, thereby posing an attractive alternative for high-dimensional queries.

The isomorphic relationship of inverted and bitmap indexes allows us to treat them interchangeably in higher-level algorithmic reasoning [29]. In fact, there exist hybrid approaches

[2] The literature often uses the term "bitmap" to refer to a bit vector, i.e., a sequence of bits. We use the term "bit*map*" only in when referring to a "bitmap index," a mapping from values to bit vectors.



(a) An inverted index.

(b) A bitmap index.

Figure 2.5: Juxtaposition of inverted and bitmap indexes: two isomorphic data structures which map lookup keys to identifier sets representing unique entries in the base data.

which combine both index types in a single data structure [38].

Currently, VAST implements its algorithms with bitmap indexes only. One reason is evolutionary: we built an early prototype of VAST on top of a specific bitmap indexing library [200], which we later replaced with our own abstractions. Another reason is technical: operations from boolean algebra directly map to native CPU instructions, enabling a uniform algorithmic framework for set intersection, union, and complement. In practice, bit vectors remain in compressed form in memory, as we discuss in the next section, and bitwise operations do not require decompression to perform set operations. In the future, we plan to investigate adaptive and hybrid approaches, but a comparative study goes beyond the scope of this thesis.

2.4.3 Space-Time Trade-off

Research on inverted and bitmap indexes strives for reducing the space requirements and increasing speed—the space-time trade-off. Reducing space involves decreases the *cardinality* or *size*, i.e., the number of distinct values ("columns"), or space of the identifier sets ("rows"). At the same time, reduced space comes at the cost of time because the techniques induce extra processing cycles during lookup. In the following, we present a unified discussion of techniques which affect the space-time trade-off. Specifically, we explain *binning*, *coding*, and *compression*. Before doing so, we introduce some common notation and terminology to formalize the previously introduced concepts, which Table 2.3 summarizes.

2.4.3.1 Terminology

Let X be a finite domain of values representing the key space in index lookups. Without loss of generality, we restrict X such that $X \subseteq 2^w$ for some constant w. An *identifier set*

Notation	Description
$x \in X$	Value x from domain X
B	Bit vector: sequence of 0s and 1s
$\|B\|$	The number of 1s in B
$B[i]$	The i-th bit of B
$L \subset \mathbb{N}_0^+$	Position list
$\|L\|$	The number identifiers in L
W	Inverted index
M	Bitmap index
S	Abstract identifier set: B or L
I	Abstract index: W or M
$\|I\| = N$	Index size: total values added
$\#I = C$	Index cardinality: distinct values
$\mathbb{0}$	Identifier set with no identifiers
$\mathbb{1}$	Identifier set with all identifiers
$\langle \beta_k, \ldots, \beta_1 \rangle$	k-component base
$\langle x_k, \ldots, x_1 \rangle$	Decomposed x according to β
$K^\beta = \langle I_k, \ldots, I_1 \rangle$	Multi-component index using β
EQ	Equality query class
1RQ	One-sided range query class
2RQ	Two-sided range query class
RQ	Range query class: $1RQ \cup 2RQ$

Notation	Description
$\mathcal{E}$	Equality-encoded S
$\mathcal{R}$	Range-encoded S
$\mathcal{I}$	Interval-encoded S

Operation	Description
$A \mid B$	Bitwise OR
$A \mathbin{\&} B$	Bitwise AND
$A \oplus B$	Bitwise XOR
$\neg A$	Bitwise NOT
$A \vee B$	Logical OR
$A \wedge B$	Logical AND
$A \oplus B$	Logical XOR
$\overline{A}$	Logical NOT
$A \cup B$	Union
$A \cap B$	Intersection
$A \mathbin{\Delta} B$	Symmetric difference
$\overline{A}$	Complement

Table 2.3: Our notation to formally describe inverted and bitmap indexes.

$S \subset \mathbb{N}_0^+$ represents a finite set of identifiers which point to external base data (e.g., records, tuples, documents, events). Each value x has associated with it one identifier $\alpha \in S$, which we denote by $x^{(\alpha)}$ as needed. The representation of S can have two forms in our framework. In the context of inverted indexes, S manifests as *position list* L, which simply contains the integer values of S. In the context of bitmap indexes, S manifests as a *bit vector* B, which is an ordered sequence of 0s and 1s. The *size* $|S|$ reflects the number of identifiers in S. For positions lists, size translates to the number of elements in the list, but for bit vectors corresponds to the *count* (aka. Hamming weight, population count, or sideways sum), i.e., the number of 1-bits. For example, the equivalent identifier sets $B = \langle 100101 \rangle$ and $L = \{0, 3, 5\}$ have both size $|B| = |L| = 3$. Let $B[i]$ extract the i-th bit of a bit vector B of n bits, where $0 \leq i \leq n$. Then we can convert a bit vector into a position list by $L = \{\alpha \mid 0 \leq \alpha \leq n \wedge B[\alpha] = 1\}$. We define two special identifier sets: $\mathbb{0}$ represents the empty identifier set and $\mathbb{1}$ the complete identifier set. In terms of bit vectors, they represent a bit vector with all 0s and all 1s, respectively.

We define an *inverted index* W as a mapping from values to position lists, and a *bitmap index* M as a mapping from values to bit vectors. That is, they both represent a mapping from values to identifier sets. When the context does not require differentiation, we subsume them under *index* I that maps values to identifier sets. The index *cardinality*[3] $\#I = C$ refers to the number of distinct values in I, and the index *size* $|I| = N$ to the total number of values present in I.

The two basic primitives of an index I are adding new values and looking up existing values under a logical predicate. Conceptually, adding a new value $x^{(\alpha)}$ means the index adds α to the identifier set of x and increases $|I|$ by 1, but bumps $\#I$ only if $x \notin I$ beforehand. It is possible to adds the same value x multiple times under different identifiers, i.e., the set of all values in an index is given by $\{x^{(\alpha_i)} \mid \alpha_i \neq \alpha_j \wedge 0 \leq i \neq j < |I|\}$. After adding x for the first time, $x \in I$ holds true. A lookup produces an identifier set S describing all $x \in I$ for which a given query predicate matches. In particular, we distinguish the following classes of predicates in accordance with the literature [40]:

EQ The class of *equality queries* concerns predicates involving operators to $\circ \in \{=, \neq\}$. The point lookup $I \circ x$ returns the identifier set $S = \{\alpha \mid z^{(\alpha)} \circ x \wedge z \in I\}$. Examples of EQ-queries include $I = 42$ and $I \neq 0$.

1RQ The class of *one-sided range queries* describes inequality predicates with respect to the total ordering of the value domain X. 1RQ-queries have the same format as EQ-queries but use the operator set $\circ \in \{<, \leq, \geq, >\}$. Examples of 1RQ-queries include $I \leq 42$ and $I > 0$.

2RQ The class of to *two-sided range queries* brackets a value $x \leq I \leq y$ where $x \leq y$. A lookup yields the identifier set $S = \{\alpha \mid x \leq z^{(\alpha)} \leq y \wedge x \leq y \wedge z \in I\}$. We can express any 2RQ-query as a conjunction of two 1RQ-queries: $x \leq I \leq y \equiv x \leq I \wedge I \leq y$. An example of a 2RQ-query includes $1900 \leq I \leq 2000$.

RQ This class subsumes both 1RQ and 2RQ.

2.4.3.2 Binning

Binning reduces the index cardinality by grouping multiple values into a single one. It also discretizes continuous values, such as time or floating-point numbers. A binning function $f : X \to Z$ with $|X| \geq |Z|$ is a surjection, e.g., $f : \mathbb{R} \to \mathbb{Z}$ with $f(x) = \lfloor x \rfloor$.

However, binning introduces false positives in the lookup process, which requires a *candidate check* with the base data in order to fully resolve ambiguities. Choosing an apt binning strategy depends on the value distribution and access patterns. Basic strategies include *equi-width* binning to partition the value domain in equal-sized intervals, and *equi-depth* binning strategy to group values such that each bin has the same sum of the frequencies of occurrences of a value. More sophisticated algorithms to compute the bin boundaries can outperform basic strategies by a factor of two in terms of candidate check time [162].

[3] Graham et al. [88] denote the cardinality of a set A by $\#A$, which inspired us to adopt this notation.

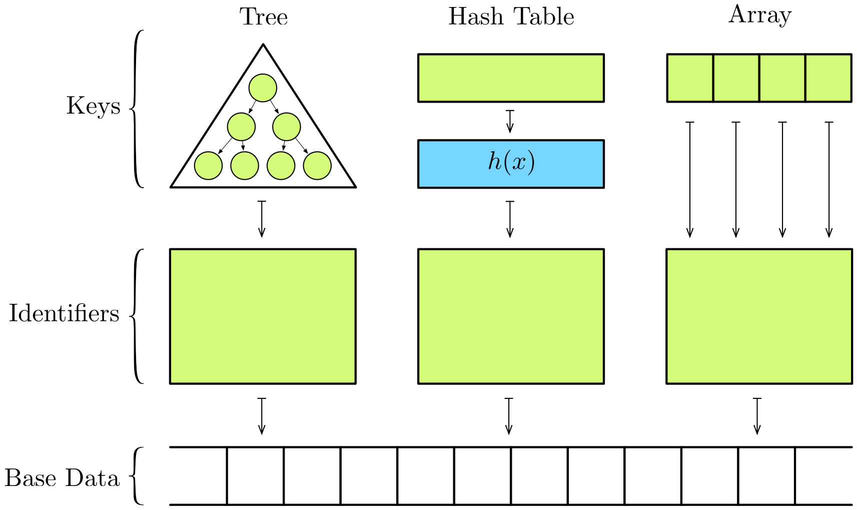


Figure 2.6: Design choices to map keys to identifier sets. Value type, access patterns, and index cardinality influence the choice of data structure (e.g., tree/trie, hash table, sorted array).

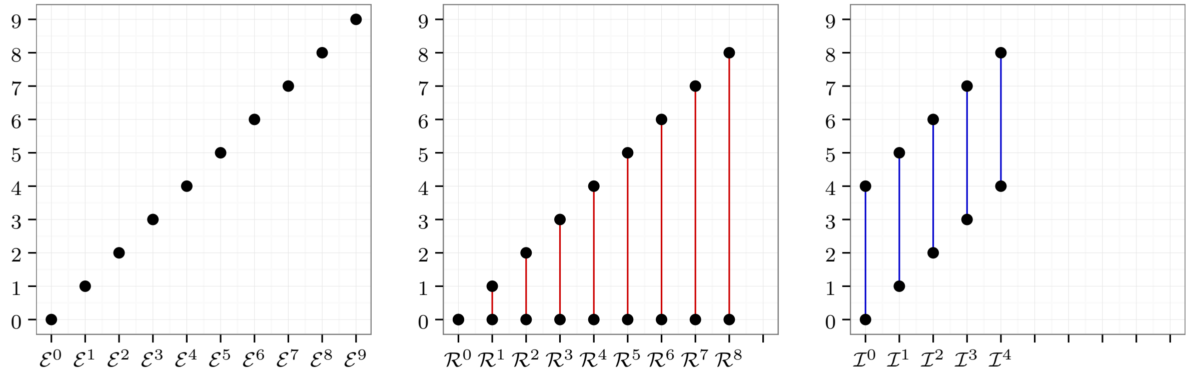
A candidate check can easily dominate the entire query execution time, due to the materialization of the additional base data (high I/O costs) and extra post-processing. Therefore, choosing an efficient binning strategy requires careful tuning and domain knowledge, or advanced adaptive algorithms. A more robust method relies on constructing multiple indexes in different resolutions [172], and then performing a lookup over multiple levels.

2.4.3.3 Coding

The *coding* scheme determines how an index incorporates new values (*encoding*) and looks up existing ones (*decoding*). Two aspects of coding exist, one concerning the mapping of values to identifier sets and the other the representation of the identifier sets themselves.

In general, mapping values to identifier sets requires an associative data structure. Figure 2.6 shows several exemplars: trees, hash tables, or arrays. Only trees and sorted arrays preserve the order of values, which can prove useful when it comes to executing *meta queries* on keys to select specific subsets of identifier sets, e.g., in range queries, when accessing all identifiers with a common prefix, or in similarity queries.

In the following discussion, we only focus on sorted arrays to access identifier sets, because they provide constant-time access by establishing a 1:1 relationship between array indices and each $x \in X$. At first, this approach may seem naive because it becomes quickly prohibitive for high index cardinalities, but binning and multi-component indexes (see below) render this an advantageous approach, because it allows for ignoring the cost of identifier set retrieval and supports all lookup operations due to the total ordering.



(a) Equality. **(b)** Range. **(c)** Interval.

Figure 2.7: Illustrating how different coding schemes associate values with identifier sets for $C = 10$: equality coding maintains one identifier set per value, range coding $C - 1$ identifier sets representing a half-open range, and interval coding uses $\lceil \frac{C}{2} \rceil$ identifier sets covering $[x, x + \lceil \frac{C}{2} - 1 \rceil]$. We give give a concrete example in Figure 2.8.

To represent identifier sets, the literature distinguishes three main schemes, which we illustrate in Figure 2.7:[4]

Equality Coding. This scheme associates each value with exactly one identifier set. The index I consists of C identifier sets $I = \{\mathcal{E}^0, \dots, \mathcal{E}^{C-1}\}$. Encoding a value $x^{(\alpha)}$ involves appending α to $\mathcal{E}^x$. EQ-queries take place in constant time, but RQ queries require merging multiple identifier sets. The first row in Table 2.6 shows the lookup algorithms for equality decoding.

Range Coding. This scheme associates each value with a range of identifier sets. The index I consists of $C - 1$ identifier sets $I = \{\mathcal{R}^0, \dots, \mathcal{R}^{C-2}\}$. Encoding a value $x^{(\alpha)}$ involves appending α to all $\mathcal{R}^i$ where $i \leq x$. Without loss of generality, we can always assume operator $\leq$, because the following identities allow for computing a lookup under the remaining inequality operators [39]:

$$I < x \equiv I \leq x - 1 \tag{2.1}$$

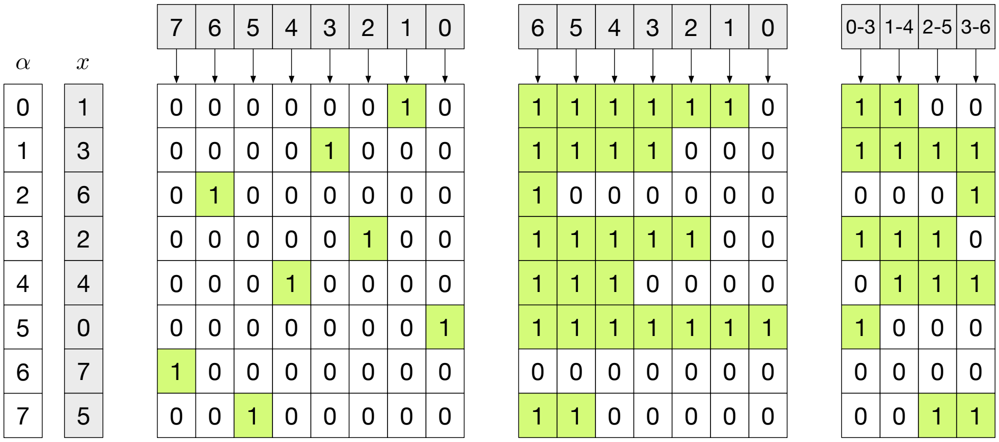
$$I > x \equiv \overline{I \leq x} \tag{2.2}$$

$$I \geq x \equiv \overline{I \leq x - 1} \tag{2.3}$$

The second row in Table 2.6 shows the lookup algorithms for range decoding.

Interval Coding. This scheme splits the index into overlapping slices, each of which covers half of the value space. The index I consists of $\lceil \frac{C}{2} \rceil$ identifier sets $I = \{\mathcal{I}^0, \dots, \mathcal{I}^{\lceil \frac{C}{2} \rceil}\}$,

[4] Hybrid schemes further combine these schemes: equality-range, OREO, and equality-interval coding [40].



α	x	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	1	0
1	3	0	0	0	0	1	0	0	0
2	6	0	1	0	0	0	0	0	0
3	2	0	0	0	0	0	1	0	0
4	4	0	0	0	1	0	0	0	0
5	0	0	0	0	0	0	0	0	1
6	7	1	0	0	0	0	0	0	0
7	5	0	0	1	0	0	0	0	0

Equality

α	x	6	5	4	3	2	1	0
0	1	1	1	1	1	1	1	0
1	3	1	1	1	1	0	0	0
2	6	1	0	0	0	0	0	0
3	2	1	1	1	1	1	0	0
4	4	1	1	1	0	0	0	0
5	0	1	1	1	1	1	1	1
6	7	0	0	0	0	0	0	0
7	5	1	1	0	0	0	0	0

Range

α	x	0-3	1-4	2-5	3-6
0	1	1	1	0	0
1	3	1	1	1	1
2	6	0	0	0	1
3	2	1	1	1	0
4	4	0	1	1	1
5	0	1	0	0	0
6	7	0	0	0	0
7	5	0	0	1	1

Interval

Figure 2.8: Equality, range, and interval coding exemplified using bitmap indexes with $C = 8$. Equality coding requires 8 identifier sets, range coding 7, and interval coding 4 identifier sets.

> where $\mathcal{I}^x$ covers all values in the interval $[x, x+m]$ with $m = \lceil \frac{C}{2} - 1 \rceil$.[5] Encoding a value $x^{(\alpha)}$ therefore involves appending α to all identifier sets $\{\mathcal{I}^i \mid x \in [i, i+m]\}$. The third row in Table 2.6 shows the lookup algorithms for interval decoding.

Figure 2.8 illustrates the coding schemes with bitmap index examples with $C = 8$. For the equality-encoded bitmap index, the value $7^{(6)}$ corresponds to a 1-bit in $\mathcal{E}^7$ at position 6. For range encoding, the value $3^{(1)}$ results in 1-bits in all bit vectors $\{\mathcal{R}^z \mid 3 \leq z\}$ at position 1. For interval coding, the value $0^{(5)}$ has only a 1-bit in the bit vector for interval $[0, 3]$, because $0 \notin [z, z+3]$ for all $z > 0$.

Optimality. Each coding scheme tailors its structure to a specific query class. Most prominently, equality coding requires the least number of bit vector scans for EQ, whereas range coding the least for 1RQ. This spawns the natural question "which scheme is *optimal* for a specific query class?" In the following, we work with an existing notion of optimality from the literature [39]. Let $Time(\mathcal{S}', C, \mathcal{Q})$ and $Space(\mathcal{S}', C, \mathcal{Q})$ denote the time and space costs of a scheme for an index with cardinality C and query class $\mathcal{Q}$. Time costs refer to the number of identifier set accesses, and space costs to the number of identifier sets the encoding requires. A coding scheme $\mathcal{S}$ is optimal with respect to a query class $\mathcal{Q}$ for a fixed cardinality C, if there exists no other scheme $\mathcal{S}'$ such that:

1. $Time(\mathcal{S}', C, \mathcal{Q}) \leq Time(\mathcal{S}, C, \mathcal{Q})$

[5] There also exists a variation when C is odd [41].

Coding	EQ Query	1RQ Query	2RQ Query	RQ Query
	$I = x$	$I = x$	$x \leq I \leq y$	$1RQ \cup 2RQ$
Equality	✓	✗	✗	✗
Range	✓*	✓	✗	✓
Interval	✗†	✓	✓	✓

* ✓ iff $C \leq 5$.
† ✗ iff $C \geq 14$.

Table 2.4: Established optimality results for coding schemes and query classes [40]. Intuitively, a coding scheme is optimal for a given index with cardinality C and query class if no other scheme (of those compared) can dominate it with respect to both space and time costs.

2. $\mathit{Space}(\mathcal{S}', C, \mathcal{Q}) \leq \mathit{Space}(\mathcal{S}, C, \mathcal{Q})$
3. at least one of inequality (1) or (2) is strict

We summarize established optimality results in Table 2.4. Multiple optimal schemes may exist, where one strictly dominates the other in time or space, and the other dominates vice versa.

2.4.3.4 Compression

While binning reduces the index cardinality, compression reduces the size of an index by shrinking the space of the identifier sets. There exists a large body of research dealing with the compression of position lists for inverted indexes [165, 120, 145]. Because we focus on bitmap indexes in this thesis, we refer the interested reader to this literature.

Likewise, bit vector compression algorithms have received significant attention. We list notable algorithms in Table 2.5. All shown algorithms operate on the basis of run-length encoding (RLE), although there also exists a hybrid approach combining RLE with position lists [38].

Compression renders bitmap indexes a viable data structure in practice. Previous work found that inverted indexes occupy half the space of compressed bitmap indexes for some workloads, and that bitmap indexes perform worse than inverted indexes at higher cardinalities [25]. However, newer compression methods [60, 76] compress twice as well, and attribute value decomposition [39] provides an effective means to control the index cardinality. Bitmap indexes also have the advantage that computing the complement merely involves flipping a bit, unlike generating a new list of identifiers. While we see many point-wise comparisons [202, 60, 76, 93], comprehensive and complete benchmarks of *all* known algorithms remain an open research project.

Algorithm	Publication	Patent-Free
BBC [10]	1995	✗
WAH [201]	2004	✗
COMPAX [76]	2010	✗
CONCISE [52]	2010	✓
WBC/EWAH [202, 121]	2010	✓
PLWAH [60]	2010	✗
DFWAH [164]	2011	✓
PWAH [188]	2011	✓
VLC [57, 65]	2011	✓
VAL [92]	2014	✓

Table 2.5: Enumeration of bit vector compression algorithms.

When we started the implementation of VAST, we could only choose between two algorithms, EWAH [121] and CONCISE [52], because all other known algorithms at this time fell under patent restrictions, preventing us from releasing our project as free open-source software. We chose EWAH because it trades space for time: while exhibiting a slightly larger footprint, it executes faster in certain conditions [93] because it can skip entire blocks of words. In the future, we plan to assess the performance of other algorithms as well.

2.4.4 Composition

The techniques of binning, coding, and compression introduced in §2.4.3 apply to one index instance. In particular, binning helps to reduce space consumption by reducing the index cardinality, but its surjective nature introduces candidate checks. *Multi-component* indexes [199, 39] represent an orthogonal technique to reduce the index cardinality, without introducing the need for candidate checks. By splitting an index into multiple *components*, each responsible for a piece of the value, the technique achieves an exponential reduction in space by decreasing the size of the value domain by a multiplicative factor for each component.

More formally, a k-component *base* (or radix) $\beta = \langle \beta_k, \ldots, \beta_1 \rangle$ defines a decomposition scheme for a domain with $\prod_{i=1}^{k} \beta_i$ distinct values. A base is *well-defined* if $\beta_i \geq 2$ for all $1 \leq i \leq k$. We only consider well-defined bases and define the cardinality of β as $|\beta| = k$.[6] Given a fixed base, we can decompose a value x into k components $\langle x_k, \ldots, x_1 \rangle$ as a linear combination:[7]

$$x = \sum_{i=1}^{k} x_i \beta_i$$

[6] Because we only consider well-defined bases, the cardinality of a base equals the zero "norm."

[7] Unlike previous work [39], we use the same dimensionality for coefficients and base vectors to avoid off-by-one confusions.

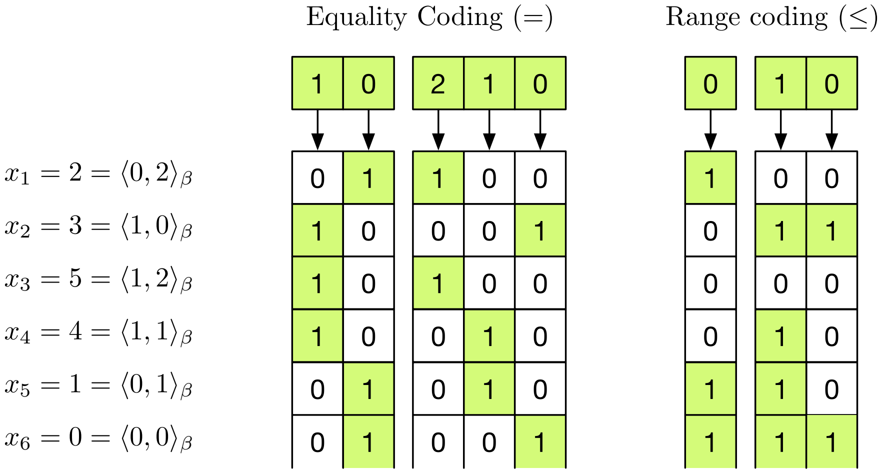

Equality Coding (=)
Range coding (≤)
1 0 | 2 1 0
0 | 1 0
$x_1 = 2 = \langle 0, 2\rangle_\beta$ 0 1 | 1 0 0 1 | 0 0
$x_2 = 3 = \langle 1, 0\rangle_\beta$ 1 0 | 0 0 1 0 | 1 1
$x_3 = 5 = \langle 1, 2\rangle_\beta$ 1 0 | 1 0 0 0 | 0 0
$x_4 = 4 = \langle 1, 1\rangle_\beta$ 1 0 | 0 1 0 0 | 1 0
$x_5 = 1 = \langle 0, 1\rangle_\beta$ 0 1 | 0 1 0 1 | 1 0
$x_6 = 0 = \langle 0, 0\rangle_\beta$ 0 1 | 0 0 1 1 | 1 1

Figure 2.9: Value decomposition according to base $\beta = \langle 2, 3\rangle$, for equality and range coding.

where

$$x_i = \left\lfloor \frac{x}{\prod_{j=1}^{i-1} \beta_j} \right\rfloor \bmod \beta_i$$

This decomposition scheme directly applies to the index structure as well: a k-component index $K^\beta = \langle I_k, \ldots, I_1\rangle$ consists of k indexes, where each I_i covers a total of β_i values. The per-component indexes can have varying coding, binning, and compression strategies according to §2.4.3. 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* [199], 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 $\Phi^k = K^\beta$ with $\prod_{i=1}^k \beta_i \leq 2^k$ as an index which supports up to 2^k values.

As an example, the value 1337 decomposes into $\langle 1, 3, 3, 7\rangle$ in base $\beta = \langle 10, 10, 10, 10\rangle$, and 42 becomes $\langle 2, 10\rangle$ in $\beta = \langle 16, 16\rangle$. Non-uniform bases allow for a mixed radix representations, as common in numeral systems for time and calendar dates. For example, a value of $x = 443,230$ seconds decomposes into "5 days, 3 hours, 7 minutes, and 10 seconds," which we can represent as $x = \langle 5, 3, 7, 10\rangle$ in base $\beta = \langle 365, 24, 60, 60\rangle$. Figure 2.9 illustrates value decomposition using bitmap indexes, for both equality and range coding according to the fixed base $\beta = \langle 2, 3\rangle$. Appending value $x = 5$, for example, requires first decomposing x into $\langle 1, 2\rangle_\beta$, and then adding each x_i to the i-th component of the index.

Choosing an apt base depends on workload distribution and query patterns. At one end of the spectrum exists the time-optimal index, which corresponds to the single-component base $\beta = \langle C\rangle$. For large values of C, this base may occupy an impractical amount of space. At the other end of the spectrum exists the space-optimal bit-sliced index Θ^k with $\beta = \langle 2, \ldots, 2\rangle$ and $k = \log_2 C$ components. This index has a much smaller footprint, but each lookup requires accessing $O(\log C)$ components. Approximative analysis of this spectrum shows that the optimal number of index components is two for the index with the best space-time trade-off [39]. However, this analysis conducted over bitmap indexes does not factor in the effect of bit vector compression, which may change the outcome.

Appending a value x to a multi-component index K^β involves two steps: first, decompose x according to β such that $x = \langle x_k, \ldots, x_1 \rangle$. Then append each x_i to index component I_i for all $1 \leq i \leq k$. Performing a lookup according to a given predicate requires a few more steps [40]:

1. **Normalization.** Before accessing the index, the normalization phase rewrites the predicate in terms of interval queries. For example, the predicate $K^\beta \in \{6, 19, 20, 21, 22, 35\}$ becomes $(K^\beta = 6) \vee (19 \leq K^\beta \leq 22) \vee (K^\beta = 35)$. After this step, we can further normalize two-sided range queries into two one-sided range queries, e.g., $(19 \leq K^\beta \leq 22) \equiv (K^\beta \geq 19) \wedge (K^\beta \leq 22)$.

2. **Decomposition.** If the predicate includes a value constant, e.g., has the form $K^\beta \circ x$, we decompose the value according to β. Thereafter, the query plan consists only of lookups at the component-level. For example, $K^{\langle 10,10 \rangle} \leq 85$ is equivalent to $I_2 I_1 \leq 8_{10} 5_{10}$ and then becomes $(I_2 \leq 7) \vee ((I_2 = 8) \wedge (I_1 \leq 5))$. The algorithm which performs this decomposition depends on the relational operator in the predicate. For equality queries we proceed as follows:

$$EQ(i, x) = \bigwedge_{j=1}^{i} (I_j = x_j) \tag{2.4}$$

This allows us to compute $K^\beta = x$ as $EQ(|\beta|, x)$, and $K^\beta \neq x$ as $\overline{EQ(|\beta|, x)}$. For one-sided range queries, decomposition depends both on the query class and the coding scheme. The algorithm *less-than-or-equal* (LE)[8] implements the multi-component range lookup as follows:

$$LE(i, x) = \begin{cases} (I_i \leq x_i - 1) \vee (\theta_i \wedge LE(i-1, x)) & i > 1, x_i > 0 \\ \theta_i \wedge LE(i-1, x) & i > 1, x_i = 0 \\ (I_i \leq x_i - 1) \vee LE(i-1, x) & i > 1, x_i = \beta_i - 1 \\ I_i \leq x_i & i = 1 \end{cases} \tag{2.5}$$

The extra parameter θ_i depends on the coding scheme and means either $I_i = x_i$ or $I_i \leq x_i$. This yields $K^\beta \leq x = LE(|\beta|, x)$, in conjunction with Equation 2.1 through Equation 2.4, a generic lookup algorithm for all relational operators $\{<, \leq, \geq, >\}$.

A small optimization can take place when all components of a value except the most significant digit have their maximum value, i.e., when $x_i = \beta_i - 1$ for all $1 \leq i < k$. For

[8] Algorithm LE with range coding is equivalent to algorithm RangeEval-Opt, which we showcase in Appendix A.

example, $K^\beta \leq 799$ with $\langle 10, 10, 10\rangle$ simplifies to $I_3 \leq 7$, which avoids two extra index component lookups $I_2 \leq 9$ and $I_1 \leq 9$.

In summary, this yields the multi-component lookup algorithm ℓ to lookup a value x in a multi-component index K^β under a relational operator $\circ$:

$$\ell(K^\beta, \circ, x) = \begin{cases} EQ(|\beta|, x) & \circ \in \{=\} \\ \overline{EQ(|\beta|, x)} & \circ \in \{\neq\} \\ LE(|\beta|, x) & \circ \in \{\leq\} \\ \overline{LE(|\beta|, x)} & \circ \in \{>\} \\ LE(|\beta|, x-1) & \circ \in \{<\} \wedge x > 0 \\ \overline{LE(|\beta|, x-1)} & \circ \in \{\geq\} \wedge x > 0 \\ \mathbb{0} & (\circ \in \{<\} \wedge x = 0) \vee (\circ \in \{>\} \wedge x = C-1) \\ \mathbb{1} & (\circ \in \{\geq\} \wedge x = 0) \vee (\circ \in \{\leq\} \wedge x = C-1) \end{cases} \tag{2.6}$$

3. **Decoding.** Finally, each component-level predicate translates into operations on encoded identifier sets. For example, $(I_3 \leq 7)$ becomes $\overline{\mathcal{E}_1^8 \vee \mathcal{E}_1^9}$ under the assumption of equality coding. The decoding step relies on the algorithms which we summarize in Table 2.6.

Coding	EQ Query $I = x$	1RQ Query $I \leq x$	2RQ Query $x \leq I \leq y$
Equality	$\mathcal{E}^x$	$\begin{cases} \bigcup_{i=0}^{x} \mathcal{E}^i & x \leq \lceil \frac{C}{2} \rceil \\ \overline{\bigcup_{i=x+1}^{C-1} \mathcal{E}^i} & otherwise \end{cases}$	$\begin{cases} \bigcup_{i=x}^{y} \mathcal{E}^i & y - x + 1 \leq \lceil \frac{C}{2} \rceil \\ \overline{\bigcup_{i=0}^{x-1} \mathcal{E}^i \vee \bigcup_{i=y+1}^{C-1} \mathcal{E}^i} & otherwise \end{cases}$
Range	$\begin{cases} \mathcal{R}^x \oplus \mathcal{R}^{x-1} & x < C - 1 \\ \overline{\mathcal{R}^{C-2}} & x = C - 1 \end{cases}$	$\begin{cases} \mathcal{R}^x & x < C - 1 \\ \mathbb{1} & x = C - 1 \end{cases}$	$\begin{cases} \mathcal{R}^x & x = y = 0 \\ \mathcal{R}^x \oplus \mathcal{R}^{x-1} & 0 < x = y < C - 1 \\ \overline{\mathcal{R}^{C-2}} & x = y = C - 1 \\ \overline{\mathcal{R}^{x-1}} & 0 < x < C - 1, \\ & y = C - 1 \\ \mathcal{R}^y & x = 0, 0 \leq y < C - 1 \\ \mathcal{R}^y \oplus \mathcal{R}^{x-1} & otherwise \end{cases}$
Interval	$\begin{cases} \mathcal{I}^0 & x = 0, m = 0 \\ \overline{\mathcal{I}^0} & x = 1, C = 2 \\ \mathcal{I}^1 & x = 1, C = 3 \\ \mathcal{I}^x \wedge \overline{\mathcal{I}^{x+1}} & x < m \\ \mathcal{I}^x \wedge \mathcal{I}^0 & x = m, m > 0 \\ \mathcal{I}^{x-m} \wedge \overline{\mathcal{I}^{x-m-1}} & m < x < C - 1, m > 0 \\ \overline{\mathcal{I}^{\lceil \frac{C}{2} \rceil - 1} \vee \mathcal{I}^0} & x = C - 1 \end{cases}$	$\begin{cases} \mathcal{I}^0 \wedge \overline{\mathcal{I}^{x+1}} & x < m \\ \mathcal{I}^0 & x < m \\ \mathcal{I}^0 \vee \mathcal{I}^{x-m} & m < x < C - 1 \end{cases}$	$\begin{cases} \mathcal{I}^x \wedge \overline{\mathcal{I}^{y+1}} & y < m \\ \mathcal{I}^x \wedge \mathcal{I}^0 & y = m \\ \mathcal{I}^x \wedge \mathcal{I}^{y-m} & y < x + m, x < m \\ \mathcal{I}^x & y = x + m, x < m \\ \mathcal{I}^x \vee \mathcal{I}^{y-m} & y > x + m, x < m \\ \mathcal{I}^x \vee \mathcal{I}^{x+1} & y = x + m + 1, x = m \\ \mathcal{I}^{y-2} \wedge \overline{\mathcal{I}^{x-m-1}} & x \geq m \end{cases}$

Table 2.6: Lookup algorithms for equality, range, and interval coding in query classes EQ, 1RQ, and 2RQ [40]. The notation conforms to §2.4.3, with the addition of $\mathbb{1}$ representing the position list with all identifiers or bit vector with all 1s.

Chapter 3

Architecture

> *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.*
>
> Antoine de Saint-Exupéry

Building a network forensics system which operates efficiently at scale involves meeting fundamental system challenges with adequate design choices. The major challenge lies in finding appropriate abstractions that fit the problem domain. In this chapter we present the architecture of VAST, our system for network forensics at scale. After having iterated through several prototypes, the design converged to a stable point. Throughout the discussion, we report our experiences and lessons learned from these iterations. We begin with a presentation of the rich-typed data model and query language in §3.1. Thereafter, we introduce key system components in §3.2 and show how to deploy them in §3.3.

3.1 Data Model

Since analyst time is a costly resource, maximizing productivity becomes a chief economic concern. When analysts can reason within their domain, without having to translate their thought processes to a narrow interface, they perform fewer context switches and operate more efficiently. In this light, we equip VAST with a data model rich in types and operations, geared towards the domain-specific idioms and workflows. In §3.1.1 we introduce the type system, and in §3.1.2 we explain the query language.

3.1.1 Type System

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

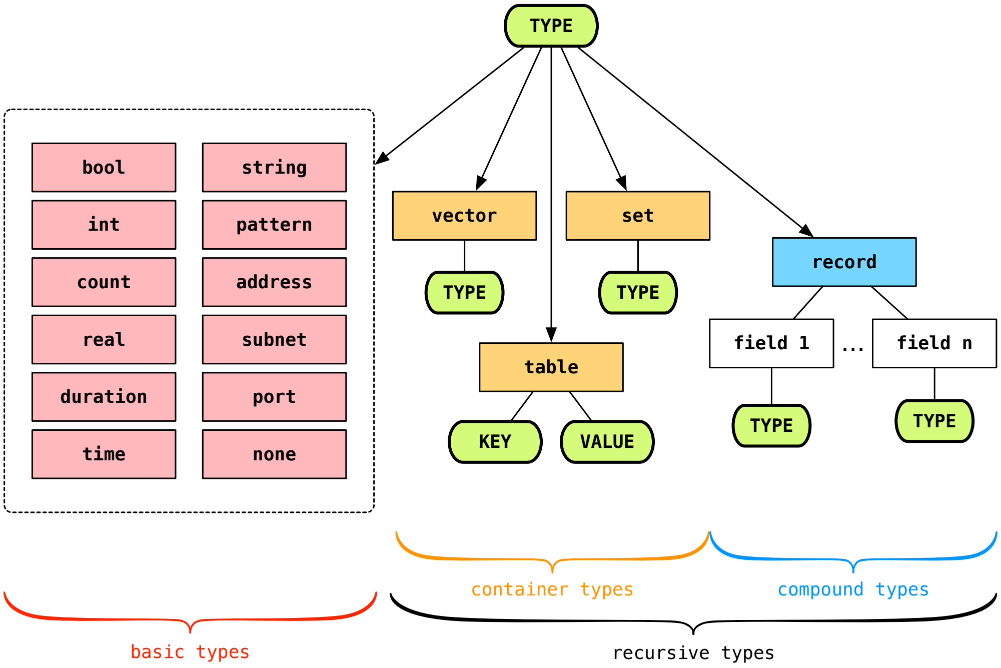


Figure 3.1: The type system of VAST, which consists of *basic* and *recursive* types. The latter split into *container* and *compound* types.

data instance. An *event* is a value with additional metadata, such as a timestamp, a unique identifier, and arbitrary key-value pairs. Figure 3.1 illustrates the type system schematically: *basic types* represent a single value (booleans, signed/unsigned integers, floating-point, times and durations, strings, IPv4 and IPv6 addresses, subnets, ports), *container types* hold multiple values of the same type (vectors, sets, tables), and *compound types* act as heterogeneous structures (records), where each named field holds a value of an arbitrary type. A *schema* describes the access structure of one or more types. Two types are *equal* if and only if they have the same signature. Two types are *congruent* if they have the same physical representation. Equality implies congruence.

As an example, consider the basic type $T = \texttt{count}$ representing a 64-bit unsigned integer. Built-in types have no name. We can create a copy $U = T$ with name $\nu(U) = \texttt{bytes}$, indicating that this type represents a number of bytes. The signatures of T and U differ since $\nu(T) \neq \nu(U)$. Hence they are no longer equal but still congruent. Let x be a value with type U and data 42. The value x becomes an event once we attach a timestamp, an ID, plus optional metadata such as $\langle \texttt{source} = \texttt{ids}, \texttt{group} = \texttt{dmz} \rangle$.

3.1.2 Query Language

VAST's query language supports filtering data according to boolean algebra. Table 3.1 lists the key syntactic elements. A query *expression* consists of one or more *predicates* connected

Boolean Expression	Symbol
Conjunction	E_1 `&&` E_2
Disjunction	E_1 `||` E_2
Negation	`!` E
Group	`(`E`)`
Predicate	*LHS* ○ *RHS*
Relational Operator ○	**Symbol**
Arithmetic	`<, <=, ==, !=, >=, >`
Membership	`in`, `!in`
Match	~, !~
Extractor (LHS/RHS)	**Semantics**
`:T`	All values having type `T`
`x.y.z`	Value according to schema
`&key`	Event metadata for `key`

Types	Examples
`bool`	`T`, `F`
`int`	`+42`, `-42`
`count`	`42`
`real`	`-4.2`
`duration`	`10ms`, `8mins`, `1h`
`time`	`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>`	`[x, y, z]`
`set<T>`	`{x, z, z}`
`table<T,U>`	`{(1,"foo"), (2,"bar")}`

Table 3.1: VAST's query language.

with boolean operators AND/OR/NOT. A predicate is a boolean function over a value in the form `LHS ○ RHS`, where ○ represents a binary relational operator. VAST supports arithmetic (`<,<=,==,!=,>=,>`), membership (`in`, `!in`), and match (~, !~) operators. Each operator has a complement, e.g., `<` pairs with `>=`. At least one side of the operator must consist of an *extractor*, which specifies the lookup aspect for a value, as follows.

Schema extractor. This extractor enables named access of types in the schema. The operator "`.`" dereferences record fields, similar to structs in C. For example, in the predicate `http.method == "POST"`, `http.method` is a schema extractor, and `"POST"` is the value of type `string` to match in the record type with name `http` and field `method`.

Met extractor. This extractor refers to event metadata. The syntax uses `&key` to reference a metadata key. There exist some predefined keys for mandatory metadata, such as `&time` for the event timestamp. For example, the predicate `&time > now - 1d` selects all events within the last 24 hours.

Type extractor. This extractor leverages 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 value or record field of type `addr`). The extractor `:T` allows any combination of built-in types or type names for `T`.

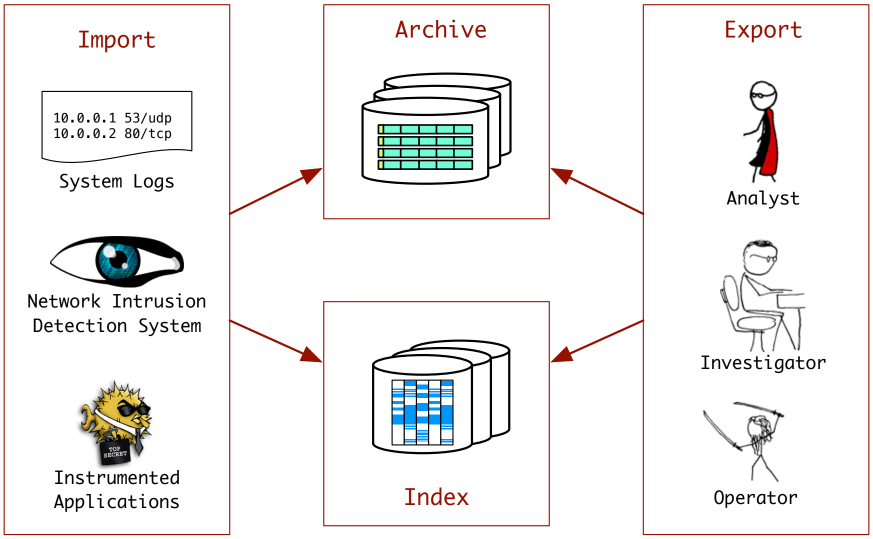
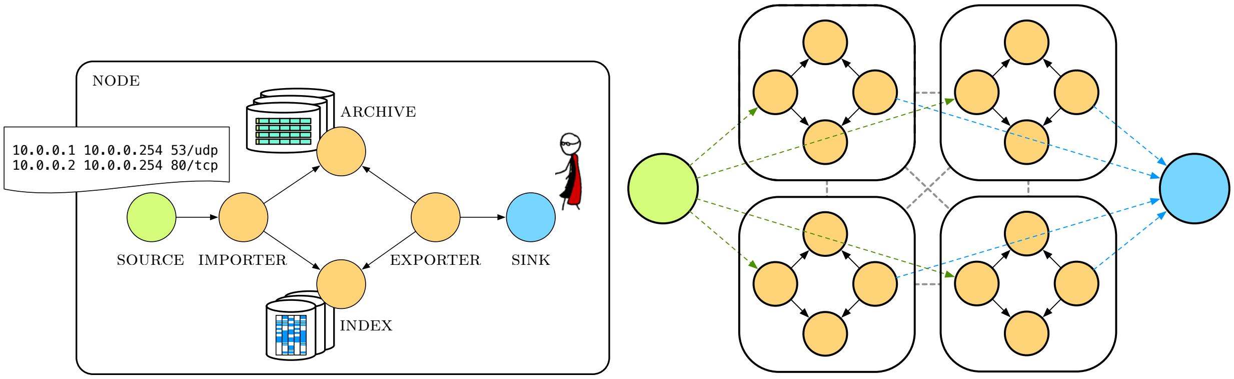


Figure 3.2: High-level system architecture of VAST showing the key components: **import**, **archive**, **index**, and **export**.



(a) Single-machine deployment.

(b) Cluster deployment.

Figure 3.3: VAST deployment styles. In a single-machine deployment, all actors run within NODE. In a cluster deployment, multiple peering NODEs form a shared-nothing system.

3.2 Components

From a high-level view, VAST consists of four key components, which we show in Figure 3.2: (*i*) **import** to parse data from various sources into events and assign them globally unique identifiers, (*ii*) **archive** to compress and store events with key-value access for retrieval, (*iii*) **index** to accelerate queries by quickly identifying the set of events to extract from the archive, and (*iv*) **export** to spawn queries and relay results to sinks of various output formats.

We model each component abstractly as a set of actors (see §2.3.1), which can execute all

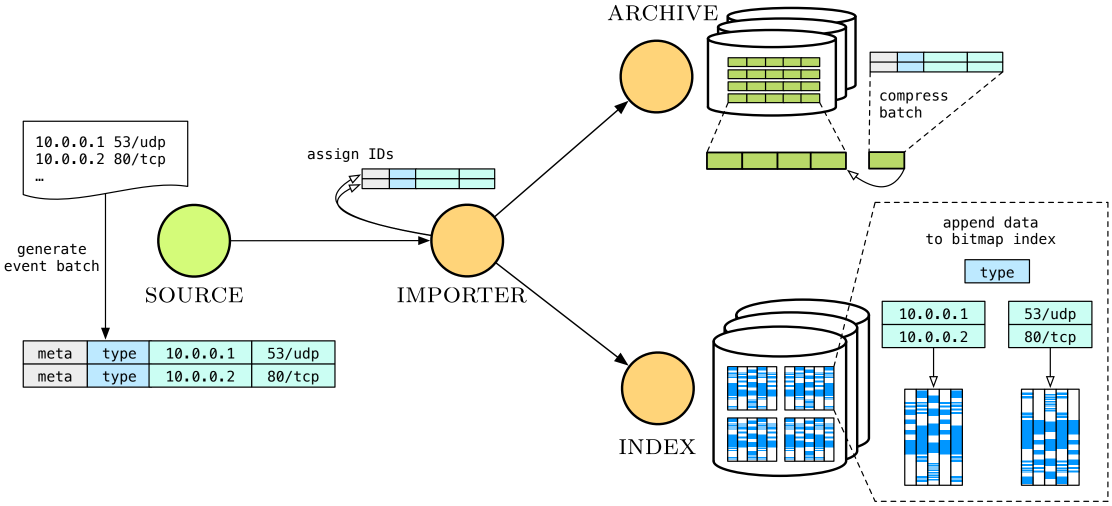


Figure 3.4: Event ingestion overview. SOURCE generates batches of events and relays them to IMPORTER. There, the events receive unique IDs before IMPORTER forwards them to ARCHIVE and INDEX.

within the same process, across separate processes on the same host, or on different machines. Unless we explicitly mention process boundaries, we assume that actors run within the same process. The fundamental actor in VAST is NODE, which acts as a container for other actors. Typically, a NODE maps to a single process. NODEs can peer to form a cluster, and then must achieve consensus over global state. VAST uses Raft [142] as its consensus algorithm, with a key-value interface to the global state state—akin to etcd [71]. We refer to this globally replicated state as *metastore*. Each NODE has access to its own metastore instance which performs the fault-tolerant distribution of values. When VAST runs in a cluster deployment, the system exhibits a shared-nothing architecture [132], where each NODE constitutes a fully independent system plus the metastore. In Figure 3.3 we show the two common deployment styles: single-machine and cluster.

Next, we present the design of the four key components in more detail. We describe in §3.2.1 how events enter the system and receive a unique ID. In §3.2.2 we discuss how ARCHIVE stores and serves events, and in §3.2.3 how INDEX processes them. In §3.2.4 we describe the event retrieval in more detail.

3.2.1 Import

The import component handles data ingestion. Various types of SOURCEs parse data into events. Once a SOURCE has completed a batch of events, it sends it to IMPORTER, where each event receives a unique identifier. Thereafter, IMPORTER relays the batch to ARCHIVE and INDEX. Figure 3.4 illustrates this process. Two notable aspects of the import process concern generating events and identifiers.

3.2.1.1 Batch Generation

A temporal or spatial condition can trigger the generation of a batch of events. For example, a SOURCE could send away its events once a second to ensure that they arrive at IMPORTER with at most a second delay. This works well for SOURCEs with very low event rates. Conversely, SOURCEs that exhibit a high event rate could cap the batch size (e.g., at 100 K events) to avoid stressing the I/O subsystem with overly large messages causing bufferbloat [114, 84].

To accommodate both low-latency and high-throughput requirements, we can combine temporal and spatial mechanisms: when event rates remain low enough, SOURCE generates a batch once per unit time. When the number of events per unit time reaches a configured upper bound, SOURCE sends the current batch regardless of the time constraint.

3.2.1.2 ID Generation

Each event represents a unique description of activity which analysts need to be able to unambiguously reference during forensic investigations. To this end, an event requires a unique identifier (ID) as metadata independent of its value. The event ID also establishes a link between the archive and index component: an index lookup yields a set of IDs, each of which identifies a single event in the archive. Given this scenario, we impose the following requirements on ID generation:

64-bit. To represent a sufficiently large number of events, we should use a pool of IDs which will not exhaust in any foreseeable future. A 32-bit ID space only yields 4 B events, which large sites easily produce within a day. Therefore, we choose a pool with an effective size of 64 bits. At the same time, we should not utilize a larger pool, because modern processors can efficiently operate on 64-bit integers and store them compactly without resorting to compound structures.

Monotonic. The indexes require monotone IDs, because they are append-only data structures which internally maintain compressed identifier sets (see §2.4.3). Consequently, they cannot efficiently support appending a value with an ID smaller than the previously inserted value. ID generation should therefore be monotonic.

Distributed. The ID generation should both support single-machine and cluster deployments. To support decentralized ID generation, some systems partition the ID into several pieces. For example, Twitter's Snowflake [111] partitions a 64-bit ID into 3 pieces: a 42-bit timestamp, a 10-bit machine identifier, and a 12-bit sequence number. However, this scheme encodes topology information into the IDs. What happens, if there ever exist more than 1024 ID-generating machines in the system? Why waste 9 bits when there exists only one machine? To avoid topology-related pitfalls, we should not encode deployment assumptions into the ID space.

```
Precondition:
    $E_i$: Event $i$ in a batch received from a SOURCE.

 1 handler RELAY($E_0, \ldots, E_{N-1}$)
 2     $[o, n) \leftarrow$ IDENTIFY($N$)    ▷ Obtain $N$ new events (see Algorithm 2).
 3     for $i = 0$ to $n - o$ do
 4         $id(E_i) \leftarrow o + i$    ▷ Assign IDs to events.
 5     end for
 6     send(ARCHIVE, $E_0, \ldots, E_{n-o-1}$)    ▷ Send the first $n - o$ events to ARCHIVE.
 7     send(INDEX, $E_0, \ldots, E_{n-o-1}$)    ▷ Send the first $n - o$ events to INDEX.
 8     if $n - o < N$ then
 9         RELAY($E_{n-o}, \ldots, E_{N-1}$)    ▷ Recurse when having received insufficient IDs.
10     end if
11 end handler
```

Algorithm 1: Assignment of IDs when IMPORTER receives a new batch of events $E_0, \ldots, E_{N-1}$ from a SOURCE.

The monotonicity requirement precludes approaches involving randomness, such as in a universally unique identifier (UUID) [117]. Moreover, any random ID generation algorithm in a space of N IDs experiences collisions after approximately $\sqrt{N}$ IDs due to the birthday paradox. In combination with the 64-bit requirement, this would degenerate the effective size of the ID space from 2^{64} to 2^{32}, and therefore not yield enough IDs. Instead, we simply choose a 64-bit counter for IDs. The first valid ID starts at 0 and the last ends at $2^{64} - 2$. The ID $2^{64} - 1$ represents the invalid (default) ID. To represent contiguous sequences of IDs, we use half-open ranges $[\alpha_0, \alpha_1), [\alpha_1, \alpha_2), \ldots$, where $\alpha_j - \alpha_i$ with $i < j$ represents the number of IDs in the respective range. To support distributed ID generation, our algorithm relies only on a single global counter in the metastore. 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 $n - o = N$ IDs.

We describe the concrete procedure in Algorithm 1, which explains how IMPORTER receives a batch of events $E_0, \ldots, E_{N-1}$, assigns IDs to them, and then forwards them to ARCHIVE and INDEX. Upon receiving N new events, IMPORTER asks a helper actor IDENTIFIER for N new IDs (line 2). We describe how IDENTIFIER handles this request shortly. The response $[o, n)$ to this request represents the range of new IDs which IMPORTER assigns to the first $n - o$ events (line 4). IMPORTER then forwards these events to both ARCHIVE and INDEX (line 6–7). If $n - o = N$, the algorithm terminates. If $n - o < N$, the algorithm recurses for the remaining $N - (n - o) - 1$ events (line 9).

In Algorithm 2 we describe how IDENTIFIER, a helper actor of IMPORTER that interacts with the metastore, handles a request for N new IDs. To avoid high latencies from frequent

Precondition:

$R_i = [l_i, r_i)$: a range of IDs, where $|R_i| = r_i - l_i$ represents the number of IDs in R_i.
$R = \langle R_0, \ldots, R_{m-1} \rangle$: a queue of m ID ranges.
B: The batch size, i.e., number of IDs to request from the metastore.
F: The factor $0 < F < 1$ indicating when to replenish.
T: The timestamp of the last replenishment.
$D_\uparrow$: The duration before increasing B
$D_\downarrow$: The duration after decreasing B
N: The number of requested IDs.

```
1  handler IDENTIFY($N$)
2      if $N > |R_0|$ then $N \leftarrow |R_0|$      ▷ Do not hand out more IDs than in the front range.
3      respond($l_0$, $l_0 + N$)      ▷ Return IDs back to sender.
4      $l_0 \leftarrow l_0 + N$
5      if $l_0 = r_0$ then $R \leftarrow R \setminus R_0$      ▷ Remove (exhausted) front range from queue.
6      if $\sum_{i=0}^{m-1} |R_i| < F \cdot B$ then      ▷ Replenish when running low of IDs.
7          if $\text{NOW} - T < D_\uparrow$ then $B \leftarrow 2B$
8          else if $\text{NOW} - T > D_\downarrow$ then $B \leftarrow B - 0.1B$
9          $R_m \leftarrow$ REPLENISH($B$)      ▷ Get $B$ new ID range from metastore.
10         $R \leftarrow R \cup R_m$      ▷ Add range to back of the queue.
11         $T \leftarrow$ NOW
12     end if
13 end handler
```

Algorithm 2: IDENTIFIER serving requests for IDs. All variables except N denote persistent state variables across handler invocations. After serving a request (line 3), IDENTIFIER replenishes its local pool of IDs if it runs below a certain fraction F of its batch size IDs (line 6).

interaction with the meta store, IDENTIFIER keeps a local cache of IDs and only replenishes it when running low of IDs. To find the right cache size, the algorithm operates adaptively (line 7–8): it doubles the cache size when replenishing twice within a fixed time interval $D_\uparrow$, and decreases the cache size when a longer time interval has passed $D_\downarrow$.

This algorithm supports both low-latency and high-throuput ID allocation due to the direct communication between IMPORTER and IDENTIFIER, the local ID cache at IDENTIFIER, and the adaptive calibration to a dynamic number of IDs that is a function of the event rate.

3.2.2 Archive

The archive holds a full copy of the raw data in compressed form. It acts as a key-value store which maps event IDs to events. During data ingestion, IMPORTER relays batches of events

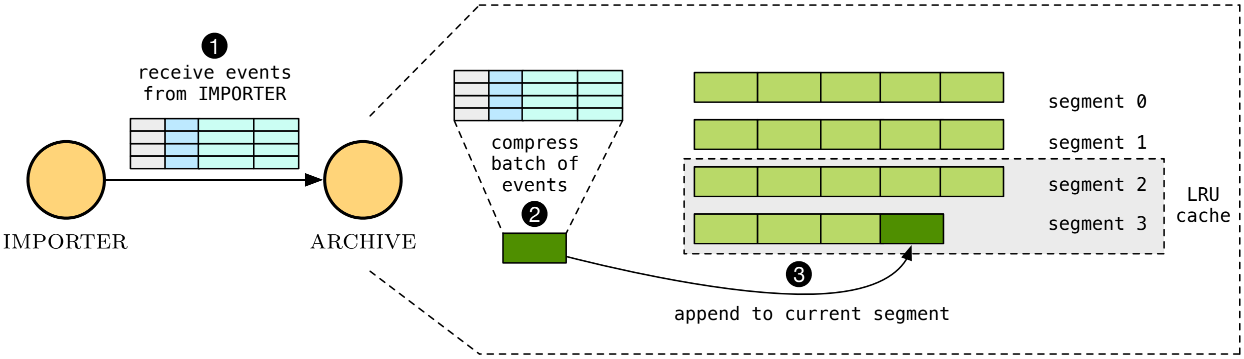


Figure 3.5: Event ingestion at the archive. When ARCHIVE receives a batch of events from IMPORTER, it compresses the batch and appends it to the current segment.

to ARCHIVE (see Figure 3.4), and for queries ARCHIVE answers lookups for specific IDs with the corresponding events (see §3.2.4.1).

To avoid frequent I/O operations for small amounts of data, ARCHIVE internally groups batches of events in compressed, fixed-size sequences (by default 128 MB) sorted by event ID. We term these sequences *segments*. When a batch of events arrives, ARCHIVE compresses the batch and appends to the current segment. Once the segment reaches its maximum size, ARCHIVE writes it to the filesystem and hands it over to an in-memory least-recently-used (LRU) cache. We illustrate this process in Figure 3.5 ARCHIVE performs I/O at the granularity of segments, as opposed to more fine-grained units like batches or individual events. The rationale for this design is two-fold: first, for a common storage architecture based on spinning disks, we want to favor sequential I/O over significantly more expensive seeks. Second, aggregating batches into segments reduces the fragmentation and overall size of the associative data structure which maps event IDs to events, as we describe next.

To map events to segments in a space-efficient way, we devised a *range map* data structure. A range map acts as an associative array with ranges as keys that additionally coalesces adjacent key ranges if and only if they have equal values. Consider the following example which illustrates how to add entries to a range map. Assume that events arrive in batches $B_i^j = [i, j)$ where i denotes the ID of the first and $j - 1$ the ID of the last event in the batch. Further, assume that a segment reaches its maximum size after 30 events. After the batches B_0^{10}, B_{10}^{25}, and B_{25}^{40} arrive at ARCHIVE, the segment consists of $S = \langle B_0^{10}, B_{10}^{25} \rangle$, but not B_{25}^{40} as it would exceed the maximum segment size. ARCHIVE records the segment in the range map under the entry $[0, 25) \to S$ and moves S into the LRU cache. Thereafter, ARCHIVE starts a new segment $S' = \langle B_{25}^{40} \rangle$ and waits until further events to arrive. In practice, the maximum segment size exceeds the batch size significantly.

ARCHIVE compresses each batch in a segment. To determine which compression algorithm works best in this scenario, we present a comparative benchmark in §5.6.1. When ARCHIVE receives a request for a specific block, it looks up the corresponding segment in the range map. If the segment does not exist in the LRU cache, ARCHIVE loads it from the filesystem and inserts into the cache. Finally, ARCHIVE delivers the batch in which the request event ID exists.

3.2.3 Index

The index accelerates search and thereby represents the cornerstone technology to enable interactive queries over large-scale data sets. We present an architecture which simultaneously supports integrating new events while answering queries. Moreover, we show how our design efficiently supports the inherently iterative workflow of network forensics.

The index transposes events from sequential values into columnar structures, while keeping a full copy of the events in the archive. This design contrasts with traditional column-stores [176, 133], which also dematerialize each event during ingestion and materialize it again later at query time. VAST opts for a sequential archive layout because it reflects the natural form of the data, allowing for easier sharing with other applications and simpler storage. In the future, we may consider switching to a more involved transposition scheme of the base data in combination with special-purpose compression schemes [78].

A scalable design supports breaking up data and compute into smaller, composable pieces. Therefore, the index consists of horizontal *partitions*, each of which hold a fixed number of events. We found that a partition size of order 1–10 M events works well in practice, which confirms previously established results [181]. In future work, we plan to perform an extensive analysis of the effect of partition sizes, and also how partition consolidation and merging can improve performance. To identify the set of relevant partitions during querying, there exists a separate *meta index*. Winnowing down the set of partitions to inspect is called *partition pruning*, and all major database systems support this feature. This entails a two-stage the lookup process: first find the set of qualifying partitions, then the set of qualifying events within each relevant partition. Since not all partitions may fit in memory at the same time, a scheduler takes care of swapping them in and out according to its schedule derived from the query expression. Figure 3.6 shows the key components of the index and the partition structure.

We first describe how to add new events in §3.2.3.1, followed by explaining how to perform lookups in §3.2.3.2. Thereafter, we sketch the meta index in §3.2.3.3.

3.2.3.1 Construction

Integrating new events must not only occur efficiently, but also asynchronously and in parallel while serving queries. To this end, we distinguish two types of partitions: *active* and *passive* partitions. Both respond to lookups, but only an active partition can receive new events. After having reached its maximum number of events, an active partition becomes passive and thereafter immutable. The INDEX maintains exactly one active PARTITION at a time. In an earlier prototype of VAST, we experimented with load-balancing events over multiple active

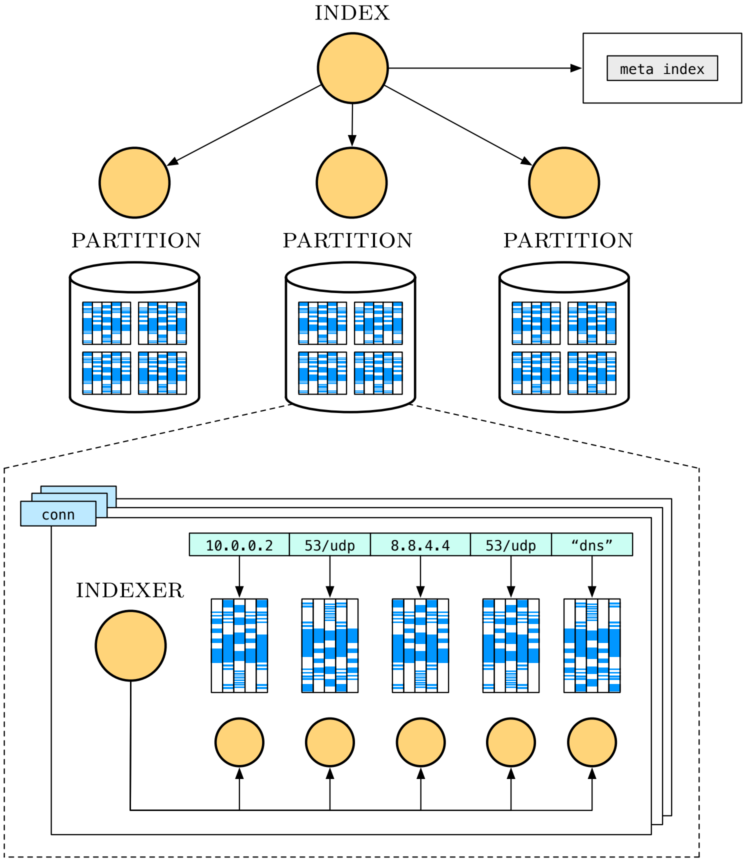


Figure 3.6: Index architecture. The INDEX consists of multiple horizontal PARTITIONs which accommodate a disjoint piece of the event data. Inside a PARTITION, there exists one INDEXER per event type, and in addition records have one helper actor per field (here: `conn` has 5 fields). The *meta index* helps during lookup with finding the relevant set of partitions, to which INDEX then forwards the query expression.

partitions, but our measurements showed that this approach does not improve performance, because each PARTITION itself offers enough potential concurrency to sufficiently parallelize the construction of indexes. That is, a PARTITION internally spawns enough actors that the runtime always fully utilizes the available hardware concurrency of the machine.

When new event batches arrive, INDEX relays them to both the meta index and the currently active PARTITION. We defer the discussion of meta index details to §3.2.3.3 and now only focus on the construction of bitmap indexes inside a PARTITION. There, each event type has its own INDEXER, which contains bitmap indexes to store the event details. For recursive record types, each indexer maintains one helper actor per field, as we illustrate in Figure 3.6 for the `conn` event. In comparison to a relational database, INDEXERs represent concurrent columns of a table.

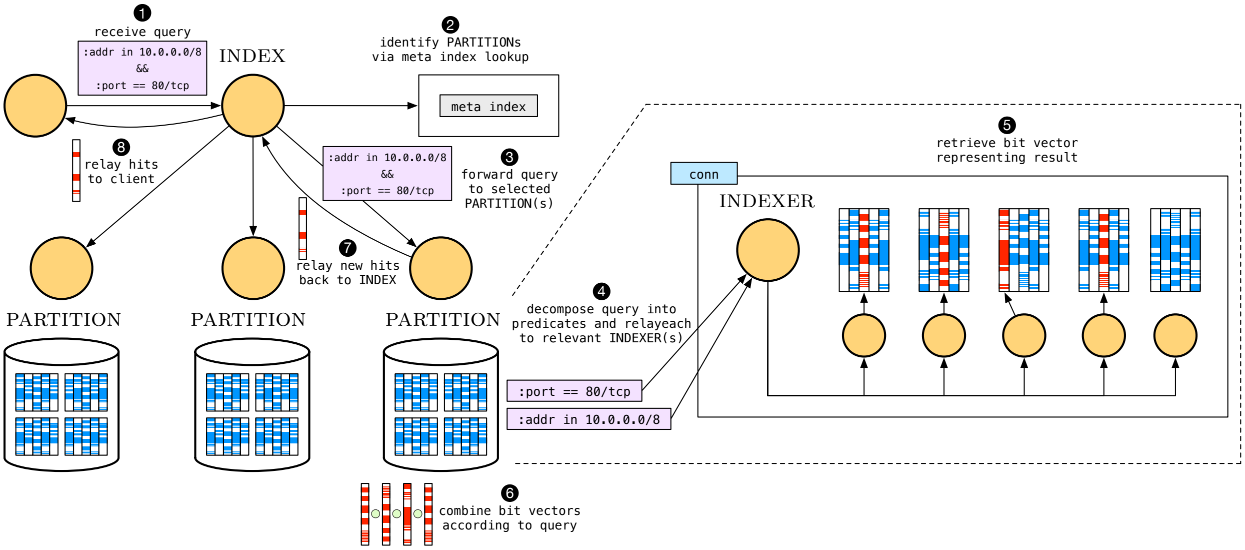


Figure 3.7: Index lookup. When INDEX receives a query, e.g., `:addr in 10.0.0/8 && :port == 80/tcp`, it identifies the relevant PARTITIONs to forward it to. Each PARTITION deconstructs the expression into its predicates and forwards them to the INDEXERs which can process the predicate. The INDEXERs performs the actual bitmap index lookup and return a bit vector to PARTITION, which evaluates the expression for each bit vector and relays new hits back to INDEX, which in turn forwards them back to the client who has requested the lookup.

In fact, PARTITION relays a batch of events concurrently to *all* INDEXERs, each of which only works on the subset matching its type. The copy-on-write message passing semantics of the actor model implementation makes this an efficient operation (see §2.3.1), because sending immutable messages to multiple receivers does not copy the message. This is a crucial property for data-intensive applications that seek a high degree of concurrency. It also resembles GPU programming: we make the data "globally" available and each execution context only operates on its relevant subset, thereby avoiding extra preprocessing and ensuring that the data exists in memory exactly once.

3.2.3.2 Lookup

The lookup process takes as input a query expression and produces as output a bit vector representing the event IDs of index hits. When INDEX receives the query expression from a client in the form of an abstract syntax tree (AST), it performs the following steps:

1. Look up the meta index to identify the set of relevant partitions (see §3.2.3.3).
2. Relay the expression to each relevant partition.

3. Receive arriving hits from partitions and forward them to the client.

When PARTITION receives the query expression from INDEX, it deconstructs the AST into predicates. The type of the predicate determines how to dispatch it further to one or more INDEXERs. Consider an example query consisting of two predicates `:port == 53/udp && foo > 42`. PARTITION sends the first predicate to all INDEXERS that manage a value of type `port`, and the second predicate to the single INDEXER responsible for the event of type `foo = count`.

When an INDEXER receives a predicate and applies it to the contained bitmap index, it responds with the bit vector representing the matching index hits. PARTITION accumulates all hits and re-evaluates the AST as it sees fit. When an evaluation triggers a change in the result for the AST, PARTITION relays the delta upstream to INDEX, which ultimately forwards it to the client.

The entire lookup operates asynchronously, incrementally, and in parallel: all loaded passive PARTITIONS process the query at the same time and send back to INDEX only the incremental *delta* of new hits since the last AST evaluation. INDEX unconditionally relays new hits from each PARTITION back to the client. This results in a continuous stream of hits, allowing the client to extract results from ARCHIVE in a pipelined and asynchronous fashion.

To accelerate lookups and provide a truly interactive experience, PARTITION maintains a *predicate cache*. This structure enables flexible recombination of predicates from already answered lookups. Moreover, it ensures efficient incremental expression refinement, i.e., when adding m new predicates to a previously answered expression of n predicates, looking up the hits for the new expressions only requires time $O(m)$. To illustrate this concept, consider the following query expressions and predicates, which we visualize in Figure 3.8:

$$\begin{aligned}
Q_1 &= A \wedge B \\
Q_2 &= A \wedge C \\
Q_3 &= (A \wedge B) \vee (C \wedge D) \\
Q_4 &= (B \wedge C) \vee D
\end{aligned}$$

Assume that a PARTITION first receives predicate Q_1, consisting of the two predicates A and B. Upon receiving Q_2, PARTITION must only retrieve C and evaluate the AST. After answering Q_3, PARTITION can answer Q_4 directly from the cache.

3.2.3.3 Meta Index

When INDEX receives a query, it first determines the set of relevant partitions by consulting the meta index. This index has different requirements than the event indexes within a partition, where each insert operation references a distinct event. For the meta index, each insert operation references the same partition until the index rotates the active partition. Furthermore, the meta index always resides in memory. To better understand the space

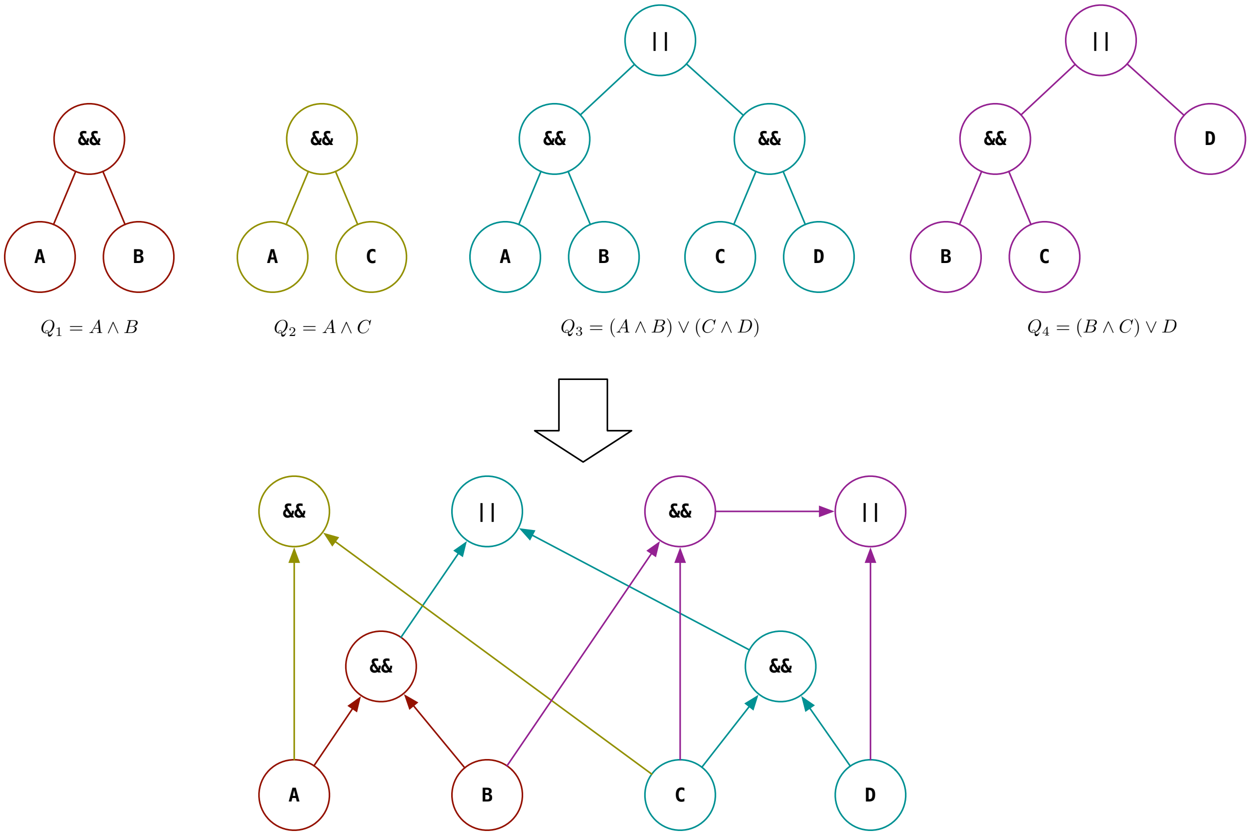


Figure 3.8: Predicate cache at a PARTITION. When adding m new predicates to a previously answered expression of n predicates, looking up the hits for the new expressions only requires time $O(m)$.

contraints, consider this example. By default, a partition has a capacity of 1 M events. Assuming that one INDEX instance can accommodate 10 B events with a given hardware setup, the meta index contains 10 K entries. The amount of budgeted memory for the meta index places an upper bound on how big a single entry can grow. With a 1 GB memory budget for the entire meta indexes, no entry can grow beyond 100 KB.

In the following, we outline the different aspects of an entry in the meta index.

Timestamp. Many queries restrict the search to a specific point or window in time. For example, the predicate `&time > now - 1h` filters all events within the last hour. We track the window of all events within a partition by keeping track of the first and last event timestamp.

Type. When a query includes a schema or type extractor, we want to be able to tell whether a partition contains events where this schema applies or whether it contains events of the given type. For example, given an event of type `foo = record{i: int, p: port}` and

Type	Index
`bool`	Two bits
`int`, `count`, `real`	Tree & binning (see §2.4.3)
`duration`	Tree & binning (see §2.4.3)
`time`	Intervals
`string`	Tokenization (see §4.2.5) & Bloom filter
`addr`, `subnet`	Radix tree
`port`	Bit vector
`vector[T]`, `set[T]`	Recursive over `T`

Table 3.2: Potential data structures for meta indexes over event data.

query `foo.i == 42 || :port == 53/udp`, the meta index lookup would tell whether a partition contains an event where the schema extractor `foo.i` and the type extractor `:port` applies. To this end, a meta index maintains a set of distinct types.

Metadata. In addition to `&time`, an event can include other key-value metadata, e.g., `&source = "dmz"` to highlight where the event came from. In practice, the number of unique values does not grow very large, and we can simply store their hash digests. When a key contains a value with higher cardinality (e.g., an IP address value, as in `&sensor = 10.1.1.42`), we can employ probabilistic data structures to summarize the set of associated values, such as Bloom filters [26]. This restricts the lookup operation to membership tests, which we deem acceptable for event metadata, since they often represent opaque tags. These data structure can generate false positives during lookups, which would result in loading a partition that does not contain events with the looked up meta data. This procedure only increases resource consumption (and perhaps latency), but does not affect correctness of the query result.

For example, if we restrict the size of a Bloomfilter to $m = 10{,}000$ bits and are willing to tolerate a false positive probability of $\phi = 0.01$, then we can compute its capacity κ, (i.e., the maximum number of elements the Bloom filter supports while maintaining ϕ during lookup) as follows:[1]

$$\kappa = \left\lfloor -\frac{m}{\ln\phi}(\ln 2)^2 \right\rfloor = 1{,}043$$

Values. The previous three aspects concern event metadata. When receiving a query such as `:addr in 10.0.0.0/8`, we would like to not only figure out whether the partition includes an event with type `addr`, but also whether there exists an address value in the given subnet. Since each type comes with type-specific query operations, the corresponding index must support them as well. For example, numeric types support

[1] This assumes that we choose the optimal number of hash functions $k^* = \frac{m}{n}\ln 2$.

inequality search, unlike addresses and subnets, which instead require top-k prefix search. Table 3.2 lists appropriate candidate data structure for each type.

For `bool`, two bits suffice: one to capture the presence of `T`, and one to capture the presence of `F`. For arithmetic types (`int`, `count`, `real`, `duration`), trees keep an ordering on the values, and binning (see §2.4.3) helps to control the value cardinality. For `time`, an interval representing the minimum and maximum point int time encountered works well. For `string`, tokenization (see §4.2.5) and Bloom filters can prove a good fit. For `addr` and `port`, radix trees support top-k prefix search. For `port`, a bit vector efficiently captures the presence of all 65,636 ports.

VAST currently only supports indexes over event metadata; we defer the implementation of value-centric meta indexes to future work. A major challenge concerns sustaining high insertion rates for tree indexes, which we need for types supporting range queries.

3.2.4 Export

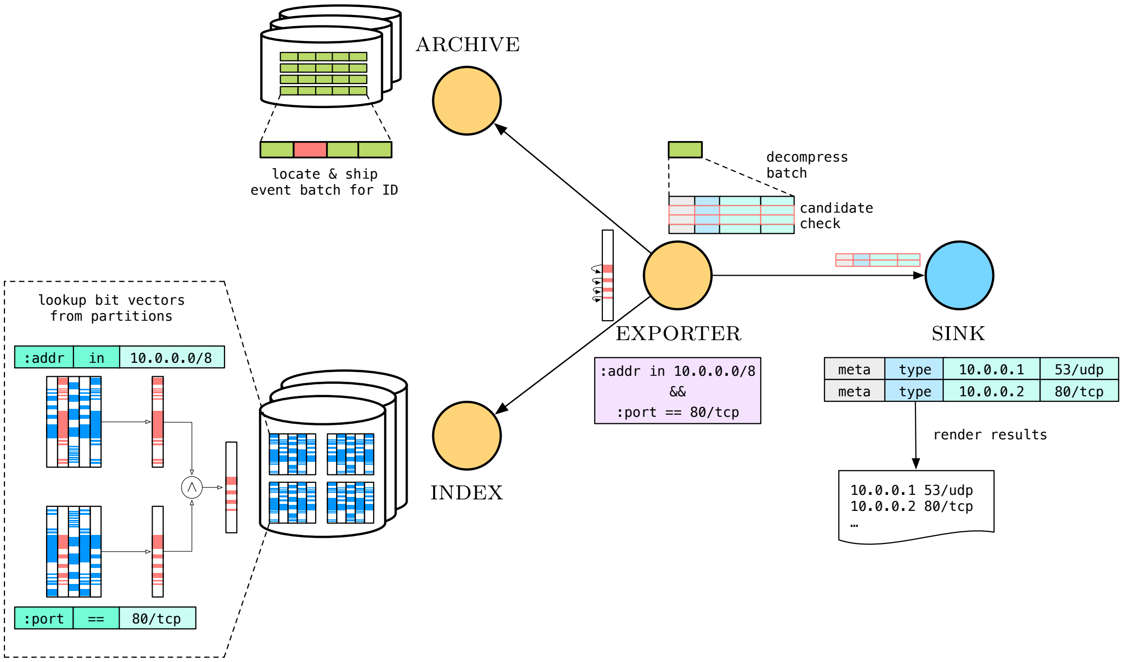
The export component implements lookup and retrieval of events, and in that sense acts as the dual to the import component. VAST spawns one EXPORTER per query that acts as the liaison between archive and index for historical queries, or as the receiver of a stream of events for continuous queries. An EXPORTER sends its result to SINKs, which perform sink-specific operations on the event stream, such as rendering to the console, writing it to a file, or relaying it via the network. VAST currently supports ASCII, JSON, PCAP, Bro [30] ASCII log, and Kafka [108] SINKs.

We begin with walking through the process of issuing historical queries in §3.2.4.1 and then discuss continuous queries in §3.2.4.2.

3.2.4.1 Historical Queries

Historical queries execute over existing data. We illustrate this process in Figure 3.9. When a VAST NODE receives a query, it spawns an EXPORTER which handles processing of this particular query instance. EXPORTER first parses the query string into an AST, and then performs normalization passes, such as moving extractors to the left-hand side of a predicate and then transforming the AST into negation normal form (see §4.3.1). Then, EXPORTER sends the normalized AST to INDEX, which processes the lookup request according to §3.2.3.2. As the stream of index hits arrives back at EXPORTER, it translates the bit vector into numeric event IDs to ask ARCHIVE for the corresponding raw events. When EXPORTER receives the batch of events, it may perform a candidate check to filter out false positives, which can occur due to surjective bitmap indexes (e.g., when using binning for floating point values). Finally, EXPORTER sends the qualifying results back to SINK. The process terminates after EXPORTER has no more unprocessed hits.

A candidate check may consume a non-negligble amount of time and may even dominate query processing [172]. We can reduce the cost either by reducing the index false positive



```
10.0.0.1 53/udp
10.0.0.2 80/tcp
…
```

Figure 3.9: Historic query architecture. EXPORTER sends the query `:addr in 10.0.0/8 && :port == 80/tcp` to INDEX, which returns a bit vector corresponding to the hits. EXPORTER traverses the bit vector and asks ARCHIVE to retrieve the corresponding batches of events. After performing a candidate check, it sends qualifying results to a SINK, which processes the events further by rendering them to the screen.

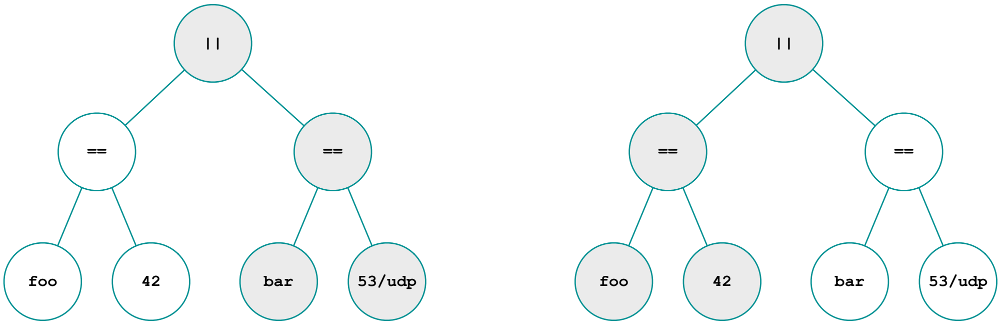


Figure 3.10: Candidate check optimization. Consider the two events `foo = count` and `bar = int`, and the query expression `foo == 42 || bar == 53/udp`. For event `foo`, the candidate checker prunes the right half of the tree (left AST), and for event `bar` the left half (right AST). This minimizes the candidate check operations per event.

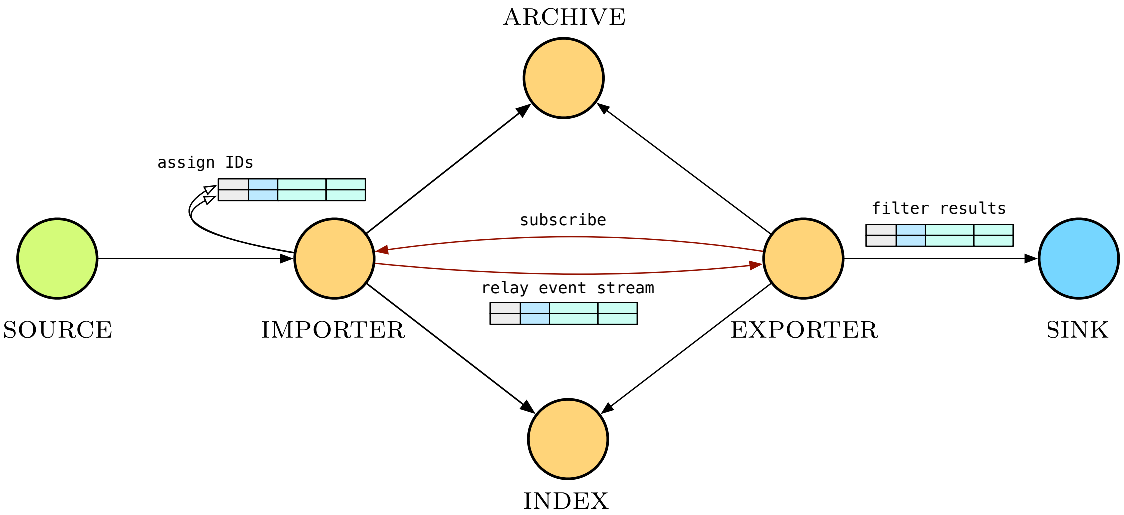


Figure 3.11: Continuous query architecture. There exists one EXPORTER per query that can serve both historical and continuous requests. For continuous queries, EXPORTER subscribes to the full event feed at IMPORTER and locally filters out the matching events.

rate or by reducing the time spent performing the check. Once EXPORTER receives the events from ARCHIVE, it can only improve on the candidate check. To this end, EXPORTER derives a unique AST for each event type from the query expression. For example, a query expression of the form `foo == 42 || bar == 53/udp` for two events `foo = count` and `bar = int` yields two different candidate checkers: one for `foo` with only the first predicate and for `bar` with only the second. We illustrate this in Figure 3.10: the left AST shows the candidate checker for event `foo`, which does not include the shaded nodes, since they only apply to event `bar`. Similarly, the right AST shows the candidate checker for event `bar`, with the nodes involving event `foo` pruned. Without this pruning optimization, each event would have to be checked against the entire AST, incurring much more operations than necessary.

Result extraction occurs in a pull-based fashion: EXPORTER only interacts with ARCHIVE when instructed to do so, e.g., when a human sink requests extracting more results. Such a design avoids performing unnecessary work and frees available resources for other tasks. It also caters to the use case when analysts only want to see a “taste” of the result, as opposed to the full set of associated activity.

3.2.4.2 Continuous Queries

VAST unifies historical and continuous queries in one architecture. The need for continuous queries arises when analysts have identified the query to describe an issue and would like to receive notification if the same activity occurs in the future.

Figure 3.11 displays the continuous query architecture. The same EXPORTER which answers historical queries can also process continuous queries by subscribing to the full event feed

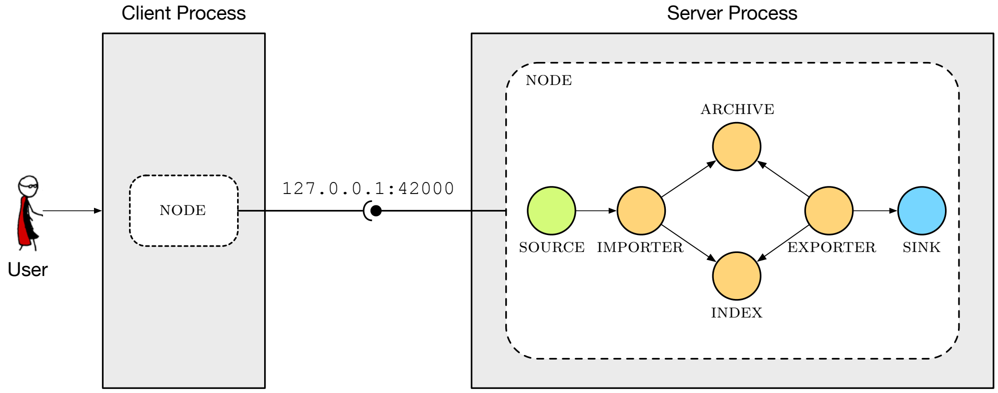


Figure 3.12: Client-server deployment. A NODE runs in a daemon process and publishes itself at a TCP endpoint. By connecting to this endpoint, client processes obtain an actor handle for the remote NODE in order to control it, e.g., based on command-line arguments, through an interactive shell, or by sending messages to it programmatically. This remote actor behaves just like a local actor, but the actor runtime transparently forwards messages over the TCP connection.

at IMPORTER. Because IMPORTER and EXPORTER run in the same process, a subscription effectively represents a "tap" into the event stream. IMPORTER only *shares* the event feed with EXPORTER. This allows to attach numerous EXPORTERs to the same feed, each of which execute concurrently. At the implementation level, this sharing translates into copy-on-write message passing: IMPORTER only forwards a pointer of the immutable data to the running EXPORTERs. However, filtering out the matching events does consume CPU cycles. To minimize the amount of work EXPORTER must perform per event, we rely on the same efficient construction of candidate checkers as for historical queries.

3.3 Deployment

So far we described VAST's main components from a conceptual perspective, without emphasizing concrete physical deployment styles. In §3.2 we mentioned single-machine and cluster deployment styles, without going into details. In §3.3.1, we take a closer look at various options of mapping components to processes and machines. Thereafter, we sketch in §3.3.2 how a cluster deployment can achieve fault tolerance.

3.3.1 Component Distribution

Users spin up a VAST instance by launching a daemon process that spawns a NODE, which consists initially of ARCHIVE, INDEX, and IMPORTER actors. The actor runtime publishes NODE at a given TCP endpoint, e.g., `127.0.0.1:42000`. When users connect to this endpoint, they obtain a remote actor handle to NODE, which gives them the opportunity to send it

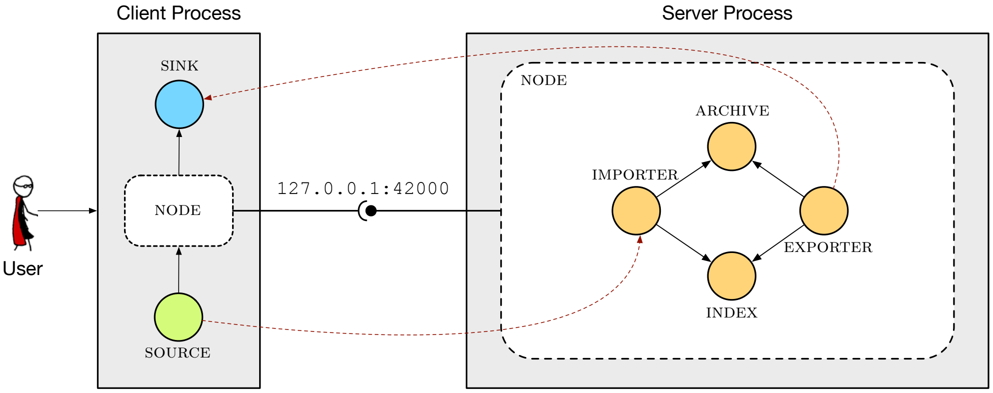


Figure 3.13: Variation of client-server deployment for interactive processing. As in Figure 3.12, a client connects to a server node through a TCP connection. The difference lies in component placement: in order to ingest events from standard input and display query results on standard output, the corresponding components must operate in a process that the user controls interactively.

control messages. For example, a user may spawn a new JSON SOURCE to ingest a log file, or spawn a PCAP SOURCE to start reading packets from a network interface. We show this form of control in Figure 3.12. Users can also run VAST in a "one-shot" mode, where NODE terminates after ingesting a single file or answering a single query. This scenario does not involve a TCP connection.

We designed VAST in accordance to the UNIX philosophy: the command line utility `vast` can ingest data from standard input and display query results on standard output. Behind the scenes, this results in slightly different component distribution, as we show in Figure 3.13 for both ingestion and querying.[2] This shows the power of the actor model in general, and the modularity of VAST's architecture in particular. The network transparent messaging layer allows for flexibly wiring components at runtime. The flexibility also proves handy in heterogeneous clusters. When a cluster includes one machine with very fast solid-state disks, deploying an INDEX instance there can result in significant latency gains.

Even though VAST supports spinning up individual components, the more common deployment form places one NODE on each machine. Each NODE runs all core actors. This style of deployment distributes the data across all machines, which has the advantage of spreading I/O and computational load (in the best case evenly) over all available machines. We illustrate this deployment style in Figure 3.14 for a user who performs distributed ingestion. The user can choose to connect to *any* available NODE, because they all share the same metastore which contains the topology information, e.g., the number of peers and the running components. The client then enumerates all IMPORTERs by querying the metastore. After connecting them

[2] In practice, a user typically either ingests data or queries, although VAST technically supports ingestion and querying simultaneously.

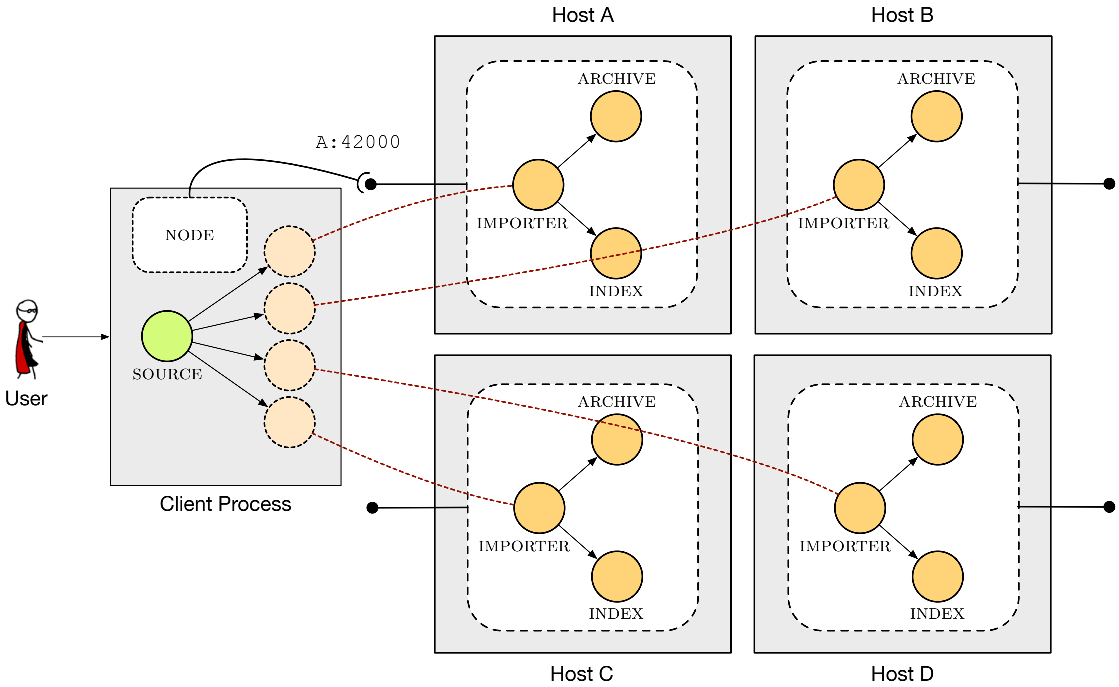


Figure 3.14: Cluster deployment showing distributed ingestion. Users can connect to *any* node in the cluster, because they all have the same data in the metastore about the topology. Ingesting a single file by default entails round-robin load-balancing of the event batches over all IMPORTERs, which the client obtains by querying the metastore.

as downstream components of the SOURCE, the ingestion process begins and load-balances the generated event batches over the IMPORTERs in a round-robin fashion.

3.3.2 Fault Tolerance

As the number of components and machines increases, the odds of failure increase as well. Large deployments routinely face broken hardware, network outages, resource exhaustion, device misconfiguration, and software bugs. Reliably operating a distributed system amidst this potential for failure requires a resilient system architecture.

Not only can hardware or software fail as a whole unit, but also at its individual components. In VAST, a SOURCE may terminate due an exception because it cannot handle a certain corner case in the input. VAST provides fine-grained, local fault isolation at the actor-level by relying on two mechanisms from the actor model: *links* and *monitors*. Two linked components have a mutual dependency. If one goes down, the other terminates as well. Monitors offer a weaker form of error propagation: if a component fails, the runtime notifies all monitors. These primitives allow for creating hierarchical supervision trees [13], independent of process or machine boundaries. To give a concrete example: a SOURCE monitors its downstream

IMPORTERs over which it load-balances its events (see Figure 3.3(b)). If one machine in the cluster fails, SOURCE will eventually receive a DOWN message after the runtime no longer has an alive connection to the machine, and as a result remove the invalid IMPORTER from its round-robin load-balancing schedule.

Comprehensive fault tolerance in a distributed setting poses a significant challenge, which goes beyond the scope of this thesis. In the following, we offer design ideas based on current trends in the field of reliable storage systems at scale.

In a distributed setting, we assume that machines come with enough "insurance" to protect against data loss, e.g., by relying on RAID or network-attached storage. While these techniques prevent data loss and durability, they do not address availability: a failed machine remains inaccessible until it comes back. *Replication* and *erasure codes* provide availability at the cost of extra storage and bandwidth during repairs [197, 161]. The simplicity of replication makes it an attractive choice in many applications in practice. For example, distributed file systems commonly replicate their data three times [85, 170, 34]. However, the massive growth in storage demands renders replication an increasingly expensive approach and spurred research on more efficient erasure codes [64, 110]. Traditional erasure codes suffer from high reconstruction costs in terms of disk I/O and network transfers (e.g., Reed-Solomon [157]). But newer erasure codes offer optimal reconstruction with respect to I/O, storage, and network bandwidth [154].

Today's systems for search perform replication at the granularity of data shards/partitions [69, 173], as opposed to at the storage layer. This gives greater flexibility when composing heterogeneous machine clusters that do not share a distributed filesystem. Conversely, relying on a distributed file system would have the advantage of (*i*) delegating the complex task to a lower layer, and (*ii*) transparently benefiting from evolution in erasure codes. In either case, a system must take care of its durable data.

VAST's durable data resides at ARCHIVE and INDEX. During failure, another NODE should take over responsibility of the data from the lost NODE. To do so, each NODE must record in the meta store (see §3.2) what slice of data it manages. Otherwise, it would remain impossible for a NODE to identify the unavailable data. ARCHIVE needs to record the IDs of the segments it accommodates, whereas INDEX the UUIDs of the partitions it takes care of. In this fashion, a NODE failing during query execution can allow another to take over its ARCHIVE and INDEX data to re-execute the query on its behalf. EXPORTER can also 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.

Data loss can still occur during ingestion, when ARCHIVE and INDEX have not yet written their data to the filesystem. We minimize this risk by writing data out as quickly as possible. When the machine crashes while writing out state, we may end up with corrupted data. For improved reliability, we could also perform opportunistic, concurrent archiving/indexing of identical data and accept the NODE that completes first as authoritative. Modern data ingestion pipelines often make use of brokers (e.g., Kafka [108]), which allow for reliable processing without introducing redundancy. Brokers offer reliability by buffering the data until receiving an acknowledgement. When consuming logs from a broker, VAST would wait until having written its state to disk before sending an acknowledgement.

3.4 Summary

In this chapter, we described the system architecture of VAST. We model the system components in terms of actors—concurrently executing entities which solely communicate via message passing—to specify the program logic and data flow independent from deployment. VAST runs on single machine, as well as on multi-machine cluster, and we showed how to replicate and compose the core building blocks to arrive at a flexible distributed system.

In particular, we described VAST's four key components: import, archive, index, and export. The import component converts data from numerous different formats into VAST's expressive data model. We explained how events enter the system and receive a unique identifier, before they get dispatched to the archive and index components. The archive acts as bulk storage for the raw data, whereas the index exists to accelerate and answer queries. The dual to the import components represents the export component, which handles query execution. We described how VAST processes historic and continuous queries in a unified framework.

Moreover, we showed how to deploy the various components over one or more machines. Only a cluster deployment can cope with the massive data volumes of large sites. The increase in machines also increases the probability of machine failure, and we ended this chapter with a discussion on fault tolerance. The actor model offers fine-grained hierarchical fault isolation, which facilitates local recovery. To ensure reliability in a multi-node scenario, we sketch how modern storage layers can provide the necessary building blocks for a more reliable distributed system.

A central theme throughout the design concerns type safety and type richness, which enable optimizations and allow users to express their data without losing crucial semantics. VAST's query language enforces strong typing, allowing users to safely use type-specific operations, such as top-k prefix search on IP addresses. In the next chapter, we pick up this topic and show how we leverage strong typing to build a composable indexing framework.

Chapter 4

Implementation

> *The problems are solved, not by giving new information, but by arranging what we have known since long.*
>
> Ludwig Wittgenstein

The architecture we describe in §3 sketches the high-level data flow between the different VAST components. In this chapter, we take a deeper look at the components' interior with a focus on implementation details that render the system an efficient platform for network forensics. In §4.1, we describe how we overcome challenges with VAST's bedrock component: the message passing layer CAF (see §2.3.2). Thereafter, we introduce in §4.2 our composable indexing framework for VAST's rich types. Finally, in §4.3 we highlight performance-critical aspects of VAST's query execution engine.

4.1 Message Passing Challenges

During our implementation of VAST, we faced two challenges that arose in combination with CAF, the actor model implementation we use for concurrency and distribution. First, we articulate the problem of overload in §4.1.1 and describe how we solved the issue with flow control. Second, we describe a message routing inefficiency in §4.1.2 and describe how we resolved it.

4.1.1 Adapting to Load Fluctuations with Flow Control

For decades, the networking community has dealt with the problem of *overload*, a situation where a sender generates messages at a higher rate than a receiver can process. As a result, the message queue at the receiver fills up until memory runs out. Protocols architect themselves out of this issue via *flow control*: sender and receiver implicitly negotiate a tractable message rate to avoid an overload scenario.

Recent work applies flow control to operating systems with QoS-inspired techniques: by ordering applications with respect to latency sensitivity, the OS can prioritize packets accordingly [90]. We consider overload from a more fine-grained perspective, within a single application consisting of components that communicate via message passing. In particular, we frame the problem in a network-transparent way: the components may reside all in the same process, within the same machine across multiple processes, or distributed over multiple machines.

The actor model [97] offers an apt vehicle to express high-level message passing independent of component deployment. Because it operates fully asynchronously, an actor can send numerous messages regardless of whether the receiver can process them. While the asynchrony enables efficient non-blocking, concurrent computation, the decoupled communication structure also allows overload scenarios to easily occur. This differs from a related model for concurrent computation, communicating sequential processes (CSP) [99], where sending a message is a synchronous operation. The receiver can block to signal overload implicitly. Per §2.3.1, the flexible failure propagation of the actor model and absence of blocking renders it a better fit for our use cases. Therefore, for the remainder of the discussion, we focus only on asynchronous flow control in the context of the actor model.

A short-sighted attempt to remedy overload provisions more buffer capacity so that the system can receive more messages. However, this ill-advised fix causes *bufferbloat* [84, 114] and ultimately worsens the situation by introducing higher latency and jitter. More buffer space does not address the root cause: a slow component on the critical path.

We approach flow control from two angles: *sensing* and *adaptation*. Sensing concerns identifying and making available the load information. Adaptation concerns changing in behavior according to a received flow-control signal. Load sensing falls into two classes:

End-to-end. When two components communicate in a request-response pattern, the round-trip time (RTT) can serve as an external measure for load. In particular, this metric works in scenarios where messages travel over multiple hops.

For example, sliding window protocols use timeouts to generate a flow-control signal. If the response does not arrive within the required latency, the sender considers the response lost and signals an overload condition.

Hop-by-hop. Regardless of the communication pattern, a component can monitor various internal performance metrics to determine its load status, such as CPU utilization, memory consumption, or message queue size.

Local introspection can yield various load signals. A continuous signal normalized to the interval $[0, 1]$ offers a range between underload and overload. One can define thresholds for underload and overload within that range, or apply control theory to incorporate feedback.

After having receiving a signal from the flow-control sensor, a component can react to it with two principle strategies:

Back-pressure. When a component senses a critical load, it must react to prevent the system from keeling over. If the component produces data locally, it can reduce its sending rate to reduce pressure downstream. If the component does not produce data itself, it must propagate the signal upstream—all the way up to the source. If it only adjusted the downstream sending rate, the load issue would still persist, but one hop closer to the source.

For example, the Reactive Streams initiative standardizes back-pressure for the Java Virtual Machine and JavaScript [155], but the concept has found wide application in the networking domain for decades [180, 192, 135, 207].

Load Shedding. When resource usage of an overloaded component grows beyond a critical point, load shedding alleviates the situation. This can mean dropping new messages to prevent the message queue from growing, skipping an intensive calculation to shed CPU load, or expunging state to free memory. Even though load shedding can affect the correctness of operation, it may prevent fatal crashes due to resource exhaustion or prevent violations of strict latency requirements.

For example, Internet routers drop packets when their queue overflows, the Bro network monitor can choose different depth of packet analysis based on its CPU load [67], and streaming databases can dynamically drop messages such that the maximum relative error across queries is minimized [14].

Load-shedding offers a simple, local solution to overload when lost data does not affect correctness of operation, e.g., when the lack of a response causes the sender to retransmit a message. But unless the sender adapts its resource usage, load-shedding only addresses the overload situation symptomatically. Conversely, back-pressure attempts to alleviate the load problem by notifying the culprit. A component only needs to know how to propagate or react to flow-control messages and adjust resource usage accordingly, whereas load-shedding only proves effective when the sender can infer from the response how to adapt.

When components run both within the same machine and across machines, the sending rate constitutes the common denominator for reacting to load changes. For example, when machine B experiences heavy CPU and memory load, and propagates overload upstream to component A on a different machine, A can only avoid sending more data downstream to alleviate the situation. Conversely, when A and B reside on the same machine, flow-control messages can contain more system-specific information and trigger a wider range of adaptations, such as

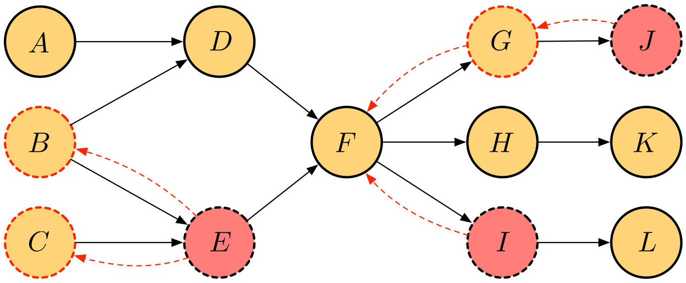


Figure 4.1: Flow control signaling. Overloaded components (E, I, J) report upstream (B, C, F, G) when they sense that they cannot keep up with the current message rate. A component which receives a flow control signal continues forwarding upstream by default, e.g., when G forwards a signal from J to F. The load-balancer F overrides this default behavior by temporarily removing G and I from the schedule.

adjusting CPU, memory, and I/O resource consumption. For the scope of this thesis, we only focus on the generic network-transparent scenario and therefore only consider adaptation of sending rate.

In VAST, components map to actors, but the actor model implementation CAF currently does not support flow control. There exist several scenarios in VAST where a sender can easily overload a receiver. Most notably, this can occur during ingestion when events arrive faster than the system can handle them. Bursts in the input rate, which can occur during denial-of-service attacks or flash crowds, exacerbate capacity planning. To prevent data loss of non-critical, latency-insensitive events, queuing brokers (e.g., Kafka [108]) act as low-pass filters to absorb peaks, but only if the ingestion capacity remains greater than the mean event rate. During ingestion, event producers (SOURCEs) can overload components downstream (IMPORTER, ARCHIVE, INDEX, and EXPORTER). To avoid overflow of the actors' mailboxes at these components, we implemented a simple yet effective back-pressure mechanism: when an actor becomes overloaded, it sends an `overload` message to all of its registered upstream components, which either propagate it further, or, if data producer themselves, throttle their sending rate. When an overloaded actor becomes underloaded again, it sends an `underload` message to signal upstream senders that it can handle more data again. This basic mechanism works well to prevent system crashes due to overloads.

To illustrate, consider the example topology in Figure 4.1, which models the data flowing between various actors. Overloaded actors (E, I, J) report upstream (B, C, F, G) when they sense that they cannot keep up with the current message rate. An actor that receives a flow control signal continues forwarding upstream by default, e.g., when G forwards signals from J to F. Assuming F has a load-balancer function and distributes every message to G, H, and I, it can override the default propagation behavior by removing G and I temporarily from the schedule, while still being able to make progress via the path to H.

We implemented this functionality by adding a new actor type to CAF. This actor keeps track

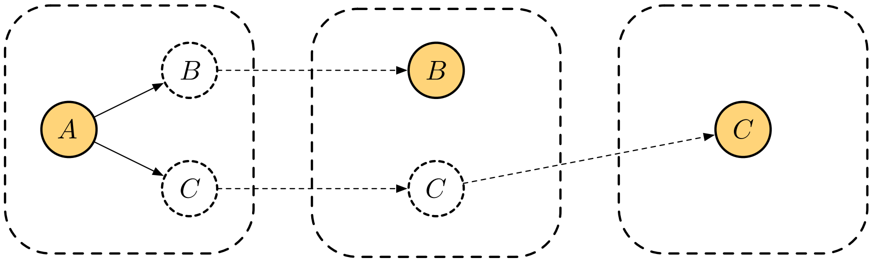


Figure 4.2: Message routing in CAF. Actors with a dashed circle represent proxy actors by the CAF runtime. When A sends a message to B, the runtime enqueues the message first in the local proxy B, which then delivers it to the node where B actually runs. In this example, A has obtained a reference to C via B. Therefore, a message from A to C travels via the node of B.

of its upstream nodes. By default, it forwards flow control messages upstream, but users can override the behavior that handles overload and underload messages, e.g., in a load-balancer scenario. We are currently working with the CAF developers to integrate flow control deeper into the CAF runtime.

4.1.2 Resolving Routing Inefficiencies with Direct Connections

During our evaluation of distributed VAST deployments, we discovered a deficiency with CAF's message routing. When deploying a system across multiple nodes, the logical topology may diverge from the physical topology, which can lead to inefficient message routing that particularly penalizes performance for data-intensive scenarios. For example, consider three actors A, B, and C, which each run on their own node node. Figure 4.2 illustrates the scenario. Actors with a dashed circle represent proxy actors by the CAF runtime, which act as a local handle to interact with the actor. There exists exactly one connection between two nodes and CAF multiplexes messages over this single connection. In the examples, there exist two connections between the left-most and middle node, and between the middle and right-most node. When A sends a message to B, the runtime enqueues the message first in the local proxy B, which then delivers it to the node where B actually runs. In this example, A has obtained a reference to C via the node in the middle. Therefore, a message from A to C spans two hops. In this design, the propagation of actor handles over the network determines the message forwarding table of each node.

This issue arises whenever an application sends an actor handle as a message to an actor on a remote node. In multi-node systems that scale by distributing load across multiple nodes, a divergence in physical and logical topology can introduce opaque bottlenecks at lower layers. In VAST, this occurs naturally in the meta store (see §3.2), the distributed key-value store for global system state. When an operator spawns a new actor on a node which peers with other nodes, the node saves the actor handle in its local meta store from where it automatically replicates to all other nodes. When a user later wants to access this actor via another node in the system, it may receive an inefficient path.

To solve this problem, we added a new *direct connection* optimization to CAF. When enabled, two nodes automatically establish a direct connection when they exchange actor handles. Internally, each CAF node contains a networking *broker*, a special actor which adds a thin layer of message framing for remote communication. We enhanced the protocol of this broker such that it supports establishing direct connections to nodes for which no direct route exists. As a result, we can now automatically create a full mesh between all nodes in a cluster and avoid inefficiencies in the underlying routing. Even though a full mesh requires a quadratic number of connections, we do not anticipate more than order hundreds of nodes participating in a VAST cluster, for which a full mesh setup work well.

4.2 Composable and Type-Rich Indexing

Until now, we glossed over the implementation details which link VAST's type-rich data model to concrete index structures. What happens once the data values reach the concrete indexes? How do we append data and retrieve results? In this section, we present our novel indexing framework, which elevates the indexes from §2.4.2 and §2.4.3 into a higher level of abstraction. In our implementation, we exclusively rely on bitmap indexes, but we keep the following discussion abstract because the concepts apply to both bitmap and inverted indexes.

The index design space we describe in §2.4.3 offers various tuning knobs with vastly different performance implications. Choosing the right set of parameters depends on the specific use case. For each type in VAST's data model, there exist different requirements derived from the supported query operations. For example, a typical operation on numeric values involves inequality queries, whereas IP addresses lookups primarily involve top-k prefix search. However, supporting inequality and top-k search requires very different index layouts and parameterization.

Before discussing layout and operations of each data type in detail, we establish some overarching framing. Recall from §3.1.1 that 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 a *value index* $\mathbb{V} = \langle N, D \rangle$ as a composite data structure consisting of a *null index* N to represent whether data is null (implemented as single identifier set), and a *data index* D, which represents a type-specific index.

A value index $\mathbb{V}$ supports the same key operations as the index from §2.4.2:

1. $\mathbb{V} \twoheadleftarrow x^{(\alpha)}$: append a value x with ID α. This operation consists of two parts: it records whether x is null, and only if non-null, stores $x^{(\alpha)}$ in the concrete data index.[1]

[1] This process resembles checking the validity of a pointer, and dereferencing it if and only if non-null.

2. $\mathbb{V} \circ x$: lookup value x under operator $\circ$. This operation retrieves the identifier set $S = \{\alpha \mid z^{(\alpha)} \circ x \wedge z \in \mathbb{V}\}$. The value index only supports the subset of relational operators $\circ$ according to its data index. For example, a range lookup fails for an IP address index because it only supports top-k prefix search.

In the following, we present the construction of data indexes for the types VAST defines. We also provide a summary of append and lookup operations of all concrete data indexes in Table 4.1 and Table 4.2.

4.2.1 Boolean Index

The boolean index $\mathbb{B}$ only operates on the two values `true` and `false`, and therefore consists of a single identifier set S. Presence in S indicates `true` and absence `false`. To add a value $x^{(\alpha)}$, we proceed as follows:

$$\mathbb{B} \twoheadleftarrow x^{(\alpha)} \;\equiv\; S \twoheadleftarrow \alpha \qquad \text{if } x = \texttt{true} \tag{4.1}$$

A boolean index only supports equality and inequality lookups:

$$\mathbb{B} \circ x \;\equiv\; \begin{cases} \overline{S} & \text{if } x = \texttt{false} \\ S & \text{if } x = \texttt{true} \end{cases} \tag{4.2}$$

When computing the complement of an identifier set $\overline{S}$, we take as reference frame the first and last ID of the index. For example, if a boolean index $\mathbb{B} = \{4, 5, 6\}$ begins at ID 2 and ends at 8, the lookup operation $\mathbb{B} \neq \texttt{false}$ yields $\{2, 3, 7, 8\}$ as result.

4.2.2 Integral Index

Integral types represent number types, such as `count` and `int`. The major challenge for such types lies in both supporting lookups using operators $\{<, \leq, =, \neq, \geq, >\}$, as well as a high cardinality value domain. To address the operator challenge, we look back to the coding schemes from §2.4.3. The operators VAST's query language supports stem from the query class $1RQ \cup EQ$, i.e., the query class of one-sided range and equality queries as in the above mentioned set of operators. For this class, both range and interval coding constitute viable candidates. We choose range coding due its simplicity and defer a thorough empirical analysis that compares range and interval coding to future work.

To address the high-cardinality challenge, we rely on multi-component indexes (see §2.4.4). How many components do we need for the 64-bit integral types `count` and `int`? To represent 2^{64} distinct values with a uniform base $\beta = \langle b, \ldots, b \rangle$, we need $\lceil \log_b 64 \rceil$ components. We choose $b = 10$ to cater to human arithmetic, which yields 20 components and thus $(10 - 1) \times$

$20 = 180$ bit vectors across all range-encoded components. (Note that in practice, unused components do not occupy space.) We briefly experimented with smaller values, such as $b = 2$ to minimize the number of identifier sets to 64. While this results in fewer identifier sets, each individual one becomes harder to compress. As mentioned above, since our focus lies on putting together an end-to-end system, as opposed to finding the optimal parameterization, we defer an in-depth empirical study of this subject to future work. An interesting aspect to explore in the future could aim for deriving an optimal base for a given value distribution, and then re-applying this base to existing indexes as well as applying it to new data.

The use of multi-component indexes introduces a complication: value decomposition according to a fixed base does not work directly with signed integer arithmetic. We cannot interpret a w-bit signed integer as an unsigned integer, because two's complement representation exhibits a different bitwise total ordering. But we can transform w-bit signed integers from $[-2^{w-1}, 2^{w-1})$ into unsigned w-bit integers using a "bias" of 2^{w-1}, which simply shifts the smallest value of -2^{w-1} to start at 0 in the unsigned representation:

$$f_w(x) = x + 2^{w-1}$$

We use this technique to specify signed integral indexes in terms of unsigned indexes. We define two indexes for unsigned and signed integral numbers: the *count index* $\mathbb{C} = \Phi^{64}$ and the *integer index* $\mathbb{C} = \Phi^{64}$ with respect to the above mentioned bias. To add a value to a count index, we compute:

$$\mathbb{C} \twoheadleftarrow x^{(\alpha)} \;\equiv\; \Phi^{64} \twoheadleftarrow x^{(\alpha)} \tag{4.3}$$

To add a value to an integer index, we compute:

$$\mathbb{I} \twoheadleftarrow x^{(\alpha)} \;\equiv\; \Phi^{64} \twoheadleftarrow f_{64}(x^{(\alpha)}) \tag{4.4}$$

The lookup follows analogously:

$$\mathbb{C} \circ x \;\equiv\; \Phi^{64} \circ x \tag{4.5}$$

And for an integer index:

$$\mathbb{I} \circ x \;\equiv\; \Phi^{64} \circ f_{64}(x) \tag{4.6}$$

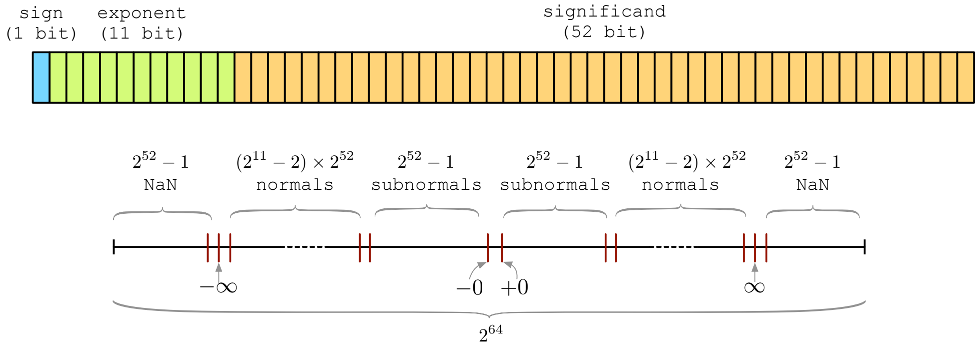


Figure 4.3: IEEE 754 double precision floating-point. The 64-bit value includes a 1-bit sign, an 11-bit exponent, and a 52-bit significand. There exist two special exponent values: 0 indicates subnormals; 2047 means ∞ if the significand (or mantissa) equals 0, and `NaN` (Not a Number) otherwise.

4.2.3 Floating Point Index

For IEEE 754 double precision floating point values [107], we use a different concept. This type consists of one sign bit s, 11 bits for the exponent e, and 52 bits for the significand m. Assembling a value from these components involves computing:

$$(-1)^s \times 2^{e-1023} \times \left(1 + \sum_{i=1}^{52} m_{52-i} 2^{-i}\right)$$

The exponent assumes a special role. With 11 bits, it can represent values in the range $[0, 2048)$. Two exponent values have a particular meaning: the lowest value of 0 indicates subnormals and the highest value 2047 means either ∞ if the significand equals 0 or `NaN` (Not a Number) otherwise. All of these special values have their negative equivalent, as we show in Figure 4.3. Per the above formula, the exponent comes with a bias of 1023. The smallest value $e = 1$ yields -1022 and the largest value $e = 2046$ yields 1023. Subnormals $(e = 0)$ fill the underflow gap around 0, and change the computation of the floating-point value as follows:

$$(-1)^s \times 2^{-1022} \times \left(\sum_{i=1}^{52} m_{52-i} 2^{-i}\right)$$

Instead of attempting to perform a similar transformation as for signed integers to map the number space into the domain of 64-bit unsigned integers, we construct a special floating-point index. This index closely follows the structure of the IEEE754 double-precision format: one identifier set for the sign, 11 for the exponent, and 0–52 for the significand.[2] Figure 4.4

[2] A variation for IEE754 single-precision works analoguously.

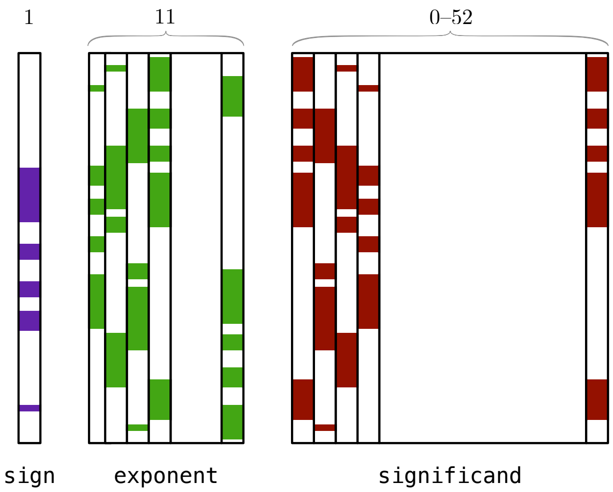


Figure 4.4: IEEE 754 floating-point index. The structure directly follows the representation in Figure 4.3, where each bit in the value maps to its own identifier set S_i.

depicts the structure of the `real` index. By allowing for a variable number of identifier sets for the significand, we get the same effect as rounding. This allows users to customize the desired precision. Instead of effectively implementing a bit-sliced index for exponent and significand, we could equally choose a multi-component bitmap index for some base β.

Let $\mathbb{F} = \langle S, E, M \rangle$ denote the floating point index for type `real` with 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. Likewise, let x_s denote the sign bit, x_e the 11-bit exponent, and x_m the 52-bit significand of x. Adding a value $x = \langle x_s, x_e, x_m \rangle$ involves computing:

$$\mathbb{F} \twoheadleftarrow x^{(\alpha)} = \langle x_s^{(\alpha)}, x_e^{(\alpha)}, x_m^{(\alpha)} \rangle \;\equiv\; \begin{cases} S \twoheadleftarrow x_s^{(\alpha)} \\ E \twoheadleftarrow x_e^{(\alpha)} \\ M \twoheadleftarrow x_m^{(\alpha)} \end{cases} \tag{4.7}$$

We define $\triangledown$ as the “mirrored” operator of $\circ$ (e.g., $<$ and $>$, or $\leq$ and $\geq$). Then, a lookup for a value x and translates into computing:

$$\mathbb{F} \circ x \;\equiv\; \begin{cases} S = x_s \wedge E \circ x_e \wedge M \circ x_m & \text{if } \circ \in \{=, \neq\} \\ S = 0 \wedge E \circ x_e \wedge M \circ x_m & \text{if } x \geq 0 \wedge \circ \in \{>, \geq\} \\ S = 1 \vee (E \circ x_e \wedge M \circ x_m) & \text{if } x \geq 0 \wedge \circ \in \{<, \leq\} \\ S = 0 \vee (E \triangledown x_e \wedge M \triangledown x_m) & \text{if } x < 0 \wedge \circ \in \{>, \geq\} \\ S = 1 \wedge E \triangledown x_e \wedge M \triangledown x_m & \text{if } x < 0 \wedge \circ \in \{<, \leq\} \end{cases} \tag{4.8}$$

We define equality lookup of two floating point values x and y as $|x - y| < \varepsilon$, where ε is determined by the precision of the significand. Unlike with exact indexes, equality does not imply transitivity.

4.2.4 Duration & Time Index

VAST represents data of type `duration` as 64-bit signed integers in nanosecond resolution. Thus, the maximum representable time span amounts to ±292.3 years. Since `duration` and `int` are congruent,[3] we express durations with an integer index. The type `time` describes a fixed point in time, which VAST internally expresses as `duration` relative to an *epoch*. We anchor time points at the UNIX epoch, January 1, 1970. Since the epoch constitutes an intrinsic part of the type, `time` is congruent to `duration`.

For both `duration` and `time`, it makes sense to explore a different value decomposition than for standard integral indexes. As we sketched in §2.4.4, when humans express time, they naturally think in hours, minutes, and seconds, as opposed to representing every duration value in seconds only. For example, the base $\beta = \langle 365, 24, 60, 60, 1000, 1000 \rangle$, which ranges from nanoseconds, milliseconds, seconds, hours, days, to years, fits more aptly in the time domain than a uniform base 10 as in generic arithmetic.[4]

Aside from a natural fit for the domain, this mixed-radix representation brings another advantage: analysis of periodicity. For example, an infected machine may check once a minute with its backend infrastructure whether to perform a certain task, such as display an ad, or attack another machine. Assuming analysts catch an instance of this check at time `08:05:42`, they may may look at activity with that particular second value of 42, ignoring all other components of time. With a domain-specific base, it suffices to look at a single component of the underlying index, which can have an impact on query latency.

In summary, we define the *duration index* as $\mathbb{D} = \mathbb{I}$ and the *time index* as $\mathbb{T} = \mathbb{D}$. Append and lookup operations are identical to the integral index, which we defined in §4.2.2.

4.2.5 String Index

Operationally relevant intelligence often takes on the form of particular markers or patterns in strings, such as a hostname, a URL path, or an email subject. To identify these markers, analysts express their queries as equality, substring, or similarity lookups. Therefore we focus on the operators $\{=, \neq, \in, \notin, \sim, !\!\sim\}$, where the last two represent similarity search. Regular expressions certainly prove even more powerful, but supporting this form of lookup via index structures poses major challenges. Therefore, most search systems that do not scan the full data offer only substring search or limited forms of "globbing."

[3] Per §3.1.1, two types are *congruent* if they have the same physical representation.

[4] This base results in $\sum_{i=0}^{5} \beta_i = 2509$ identifier sets. To achieve a smaller footprint, one may split sub-second into 10×100 instead of 1000, yielding a total of 729 identifier sets, for example. Further base decomposition allows for trading space against time.

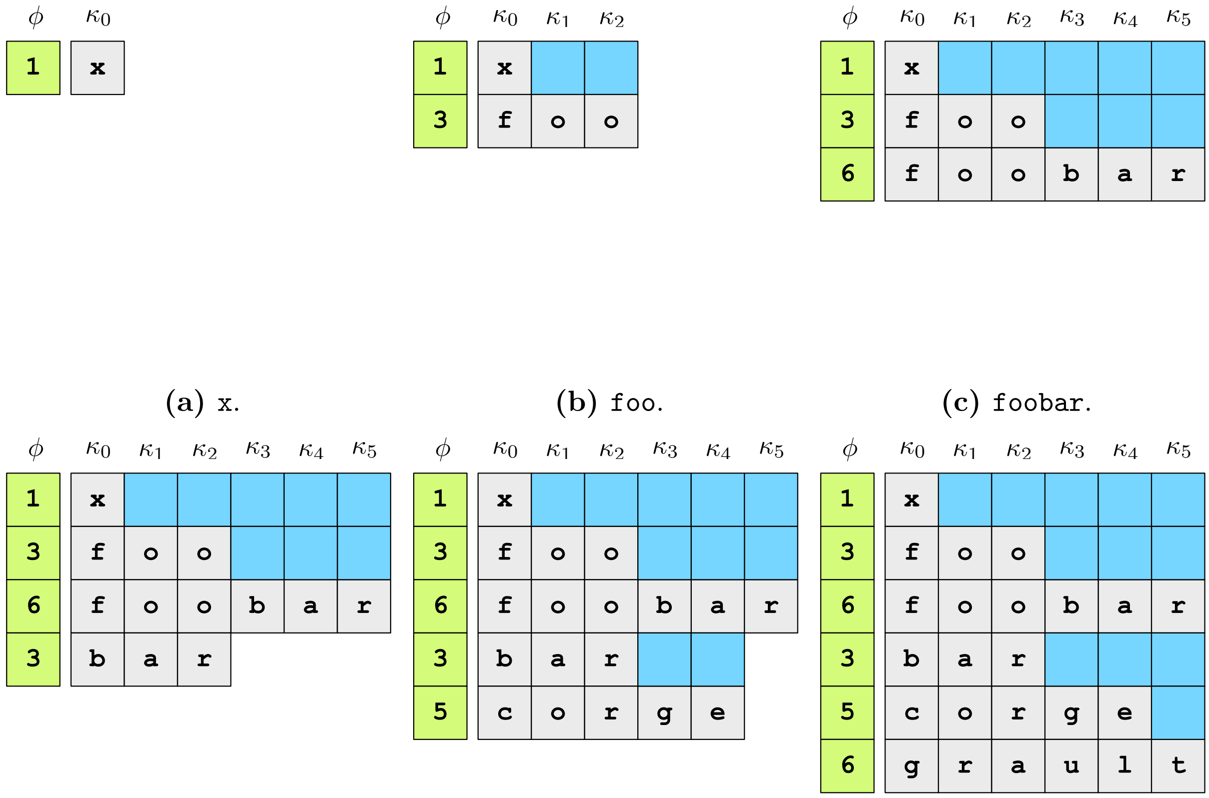
Unlike bounded numeric types, indexing strings proves more difficult due to their variable length and desired query operations. In order to leverage existing numeric index types, a common approach to string indexing involves a *dictionary* to map each unique string to a non-negative integer [174]. However, constructing a space-efficient dictionary poses a challenge in itself [131, 104]. While a dictionary-based design achieves a constant-space index structure, it only supports equality lookups: for substring search, one must search the entire dictionary key space to get a set of string identifiers, and then look up this set in the index. This takes time $O(C + h)$, where C denotes the cardinality and h the number of hits. A slight modification skips the indirection through numeric identifiers and directly attaches the keys to the identifier sets, as in Figure 2.6. That is, the identifier sets represent the codomain of the dictionary. However, this approach no longer benefits from the exponential cardinality reduction we obtain from multi-component indexes, and would require space $O(NC)$ for an index with N values, as opposed to $O(N \sum_{i=0}^{n-1} \beta_i)$. Unless the dictionary implementation relies on a hash table, it also exhibits a much lower throughput compared to the constant-time multi-component framework with numeric data.

Instead of using a stateful dictionary, another approach simply relies on a hash function to compute a unique string identifier [181]. The difference to the bijective dictionary is that two string values now may map to the same identifier, which requires a candidate check. For a hash function that produces a 64-bit digest, collisions affecting the candidate check approximately start to occur after $\sqrt{2^{64}} \approx 4\,B$ events. While space-optimal due to the absence of a stateful 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 a dictionary. Moreover, our approach can support similarity search as well. In a nutshell, our string index $\mathbb{S} = \langle \phi, \kappa_0, \dots, \kappa_M \rangle$ consists of an index $\phi = K^\beta$ for the string length, plus M indexes $\kappa_i = \Phi^8$ per character where M is largest string added to the index so far. Figure 4.5 illustrates how the index behaves when appending strings of various sizes. Each append operation adds the length of the string to the size index ϕ, and each character to the character index κ_i. For example, for the string `foo`, we have $\phi \twoheadleftarrow 3$, $\kappa_0 \twoheadleftarrow$ `f`, $\kappa_1 \twoheadleftarrow$ `o`, and $\kappa_2 \twoheadleftarrow$ `o`. More formally, we add a string value $x^{(\alpha)} = \langle x_1, \dots, x_n \rangle^{(\alpha)}$ to $\mathbb{S}$ as follows:

$$\mathbb{S} \twoheadleftarrow \langle x_1, \dots, x_n \rangle^{(\alpha)} \equiv \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \kappa_i \twoheadleftarrow x_i^{(\alpha)} \quad \forall 1 \leq i \leq n \end{cases} \tag{4.9}$$

Performing an lookup of a substring x involves computing:



(a) `x`. **(b)** `foo`. **(c)** `foobar`.

(d) `bar`. **(e)** `corge`. **(f)** `grault`.

Figure 4.5: Appending to the string index. The index consists of index ϕ for the string length, and a variable number of per-character indexes κ_i.Each panel shows how the string index grows by one new value.

$$\mathbb{S} \circ x \;\equiv\; \begin{cases} \mathbb{0} & \text{if } |x| > M \\ \phi = 0 & \text{if } |x| = 0 \\ \phi = |x| \wedge \bigwedge_{i=1}^{|x|} \kappa_i = x_i & \text{if } \circ \in \{=, \neq\} \\ \phi \geq |x| \wedge \bigvee_{i=1}^{M-|x|+1} \left(\bigwedge_{j=1}^{|x|} \kappa_{i+j-1} = x_j \right) & \text{if } \circ \in \{\in, \notin\} \end{cases} \tag{4.10}$$

Technically, the case for $\circ \in \{=, \neq\}$ is just an optimization of the generic substring case $\circ \in \{\in, \notin\}$. It only restricts the index for the number of characters in a string with an equality instead of an range ($\phi = |x|$, per Equation 4.10), which helps to speed up the evaluation by starting from a smaller set of candidates.

```
Precondition:
    $\phi$: The size index for the string size.
    $\kappa_i$: The character index for all $0 \le i < M$.
    $\circ$: A relational operator from $\{=, \neq, \in, \notin\}$.
    $x$: The string value to lookup.
    $\mathbb{0}$: The empty identifier set.
    $\mathbb{1}$: The set of all identifiers.
Postcondition:
    $R$ the identifier set according to $z \circ x$ for all $z \in \mathbb{S}$.

 1 function StringLookup($\circ$, $x$)
 2     $R \leftarrow \mathbb{0}$
 3     if ($\circ \in \{=, \neq\}$) then
 4         if ($|x| \le M$) then
 5             $R \leftarrow (\phi = |x|)$
 6             for $i = 1$ to $M$ do
 7                 $S \leftarrow (\kappa_i = x[i])$
 8                 if ($S = \mathbb{0}$) then break
 9                 else $R \leftarrow R \cap S$
10             end for
11         end if
12     else if ($\circ \in \{\in, \notin\}$) then
13         if ($|x| \le M$) then
14             for $i = 1$ to $M - |x| + 1$ do
15                 $U \leftarrow \mathbb{1}$
16                 $absent \leftarrow$ false
17                 for $j = 0$ to $|x| - 1$ do
18                     $S \leftarrow (\kappa_{i+j} = x[j])$
19                     if ($S = \mathbb{0}$) then
20                         $absent \leftarrow$ true
21                         break
22                     else
23                         $U \leftarrow U \cap S$
24                     end if
25                 end for
26                 if (not $absent$) then $R \leftarrow R \cup U$
27             end for
28             $R \leftarrow R \cap (\phi \ge |x|)$
29         end if
30     end if
31     if ($\circ \in \{\neq, \notin\}$) then $R \leftarrow \overline{R}$
32     return $R$
33 end function
```

Algorithm 3: StringLookup: looking up a value in the string index. The algorithm supports both equality lookups (line 4–11) and substring search (line 13–29).

Efficient substring search has received extensive attention in the literature. In particular, the algorithms Aho-Corasick [6], Knuth-Morris-Pratt [113], and Boyer-Moore [28] can improve over the naïve brute force. Can we apply these algorithm in our scenario? Our problem consists of matching a single substring in a set of strings encoded in bitmap indexes. Aho-Corasick attempts to solve matching multiple substrings at once in a single string. The other two algorithms attempt to locate a single substring within a single piece text. First, we have to rule out Boyer-Moore, because the algorithm requires access to original characters when computing the amount of characters to shift the substring search upon failure, but our string index design does not store the original characters. Conversely, Knuth-Morris-Pratt pre-computes a jump table as a function of the pattern only. Because Knuth-Morris-Pratt is a special case of Aho-Corasick when implemented as deterministic finite automaton (DFA) and we only consider search of single substring, we ignore Aho-Corasick for the scope of this thesis. An interesting avenue for future work explores matching several substrings in multiple strings in a single pass.

The existing algorithms all have the problem that they terminate after having located the first instance of a substring in the text. However, our string index consists of a set of encoded strings, and a lookup should return all strings where the substring matches. Consequently the algorithm must not terminate on the first match, because other matches may well appear later on. But perhaps there exists a modification to the terminating behavior that allows us to leverage some of the algorithms' key ideas to improve on our version. We are currently working on an adaption of Knuth-Morris-Pratt, but for now, our brute-force algorithm which we codify in Algorithm 3, works well enough as a proof-of-principle.

If we choose to represent 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 6-th bit determines casing in ASCII, and by simply omitting the corresponding identifier set Θ^8_6 during lookup, case-insensitive search executes *faster* than case-sensitive search.

A major challenge concerns efficient spatial representation of the string index layout. Fortunately, each append operation of a string x with $|x| = n$ only affects the first n of M indexes, and we can rely on space-efficient "vertical" representation due to compression of identifier sets. However, this index design does not lend itself well to large strings beyond a few hundred characters, e.g., when a string represents an entire HTML page. For large strings, full-text index solutions rely on tokenization in such scenarios [69, 126]. This preprocessing step splits a string according to a pattern (e.g., whitespace for text, '/' for URIs, etc.), and creates a set of multiple smaller strings, each of which to index separately. In combination with hashing and candidate check, tokenization can enable equality-lookups on substrings.

Presently, we have not yet implemented similarity search for operators $\circ \in \{\sim, !\sim\}$, but give a brief outlook how to pursue it in the presented framework. The field of information retrieval [16] deals comprehensively with this topic. One example involves *stemming* [125, 149], the reduction of words to a canonical root, e.g., `surfing` → `surf`. Our string index design retains the full information, which renders stemming an optional feature at query time. Other full-text engines [126] apply stemming upon index construction. There exist various other forms of string similarity [195], often based on *edit distance*: the minimum number of character insertions, deletions, and substitutions to convert one string to another. For example, the strings `surf` and `smurf` have edit distance 1. Applying such transformations at query time hold promise for more powerful forms of string search.

4.2.6 IP Address Index

In network forensics, IP addresses constitute a central data type to describe the endpoints of communicating entities. Since IPv4 and IPv6 naturally coexist a in today's networks, our index design must accommodate both protocol versions. A common operation on IP addresses involves 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.

Before discussing the index layout, let us review the unified representation of IPv4 and IPv6 addresses. There exists a standardized scheme to embed a 32-bit IPv4 address inside a 128-bit IPv6 address [17]: set the first 96 bits to 0 and copy the IPv4 address in the last 32 bits. To determine whether an IP address is IPv4 or IPv6, it suffices to test whether the first 96 bits equate to 0. These structure yield a natural index layout: a 128-component bit-sliced index Θ^{128} with one identifier set per IP address bit. Let $\mathbb{A} = \langle \Theta_1^{128}, \ldots, \Theta_{128}^{128} \rangle$ denote the address index with bit-slice index Θ_i^{128}, and let $x = \langle x_1, \ldots, x_{128} \rangle$ denote the bitwise representation of an IP address instance. To add x to $\mathbb{A}$, we compute:

$$\mathbb{A} \twoheadleftarrow \langle x_1, \ldots, x_{128} \rangle^{(\alpha)} \;\equiv\; \Theta_i^{128} \leftarrow x_i^{(\alpha)} \quad \forall 1 \leq i \leq 128 \tag{4.11}$$

Performing a top-k search for an IP address involves computing the intersection of the first $k > 0$ identifier sets:

$$\mathbb{A} \circ x \;\equiv\; \begin{cases} \bigwedge\limits_{i=1}^{k} \Theta_i^{128} = x_i & \text{if } \circ \in \{\in\} \\ \overline{\mathbb{A} \ni x} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.12}$$

Note that an equality lookup is a special case of prefix search for $k = 128$. Since the first 96 bits always equal to 0 for an IPv4 address, it may seem wasteful to go through them. A possible optimization includes a separate boolean index $\mathbb{B}$ to record whether an IP address is version 4, and then perform a lookup as follows:

$$\mathbb{A} \ni x \;\equiv\; \begin{cases} \mathbb{B} = \texttt{true} \wedge \displaystyle\bigwedge_{i=97}^{i+k-1} \Theta_i^{128} = x_i & \text{if } x = \texttt{v4} \wedge \circ \in \{\in\} \\ \displaystyle\bigwedge_{i=1}^{k} \Theta_i^{128} = x_i & \text{if } x = \texttt{v6} \wedge \circ \in \{\in\} \\ \overline{\mathbb{A} \ni x} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.13}$$

This speeds up IPv4 queries at the cost of a boolean index.

4.2.7 Subnet Index

In addition to IP addresses, VAST also features a `subnet` data type. This type consists of an address plus a numeric prefix. Since subnets represent aggregations of IP addresses, natural 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 $\mathbb{U} = \langle \mathbb{A}, \Phi^8 \rangle$ consists of an address index $\mathbb{A}$ and an index for the prefix. Let $x = \langle x_a, x_p \rangle$ denote a subnet value with network address x_a and prefix x_p. To add x to $\mathbb{U}$, we compute:

$$\mathbb{U} \twoheadleftarrow \langle x_a, x_p \rangle^{(\alpha)} \;\equiv\; \begin{cases} \mathbb{A}_i \twoheadleftarrow x_{a_i}^{(\alpha)} & \forall i \leq 1 \leq p \\ \Phi^8 \twoheadleftarrow x_p^{(\alpha)} \end{cases} \tag{4.14}$$

To perform a membership lookup which checks whether a given prefix x exists in $\mathbb{U}$, we compute:

$$\mathbb{U} \circ x \;\equiv\; \begin{cases} \Phi^8 \leq x_p \wedge \left(\displaystyle\bigwedge_{i=1}^{p} \mathbb{A}_i = x_{a_i} \right) & \text{if } \circ \in \{\in\} \\ \overline{\mathbb{U} \ni x} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.15}$$

Since valid prefixes lay in the small set $\{0, \ldots, 128\}$, we use a single-component equality-coded index for Φ^8, e.g., with base $\beta = \langle 129 \rangle$ to include `/0` and `/128` lookups.

While the subnet index naturally supports subset lookups, point lookups of a single address a prove more difficult, because we do not know how to mask a prior to performing a lookup, which is necessary because the subnet index only contains a network address. There exists still a method, however, to retrieve the desired result by masking a with all possible prefixes:

$$\mathbb{U} \circ a \;\equiv\; \begin{cases} \bigvee_{i=0}^{128} \left(\Phi^8 = i \wedge \left(\bigwedge_{j=1}^{i} \mathbb{A}_j = a_j \right) \right) & \text{if } \circ \in \{\in\} \\ \overline{\mathbb{U} \ni a} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.16}$$

When a is an IPv4 address and $k \in \{0, \ldots, 32\}$, we must instead perform the lookup for $k + 96$. An optimized version for an IPv4 address a^{v4} works as follows:

$$\mathbb{U} \circ a^{v4} \;\equiv\; \begin{cases} \bigwedge_{i=1}^{96} \mathbb{A}_i = 0 \wedge \bigvee_{i=97}^{128} \left(\Phi^8 = i \wedge \left(\bigwedge_{j=97}^{i} \mathbb{A}_i = a_i^{v4} \right) \right) & \text{if } \circ \in \{\in\} \\ \overline{\mathbb{U} \ni a^{v4}} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.17}$$

4.2.8 Port Index

Transport-layer ports identify the specific services that run on a machine. VAST's `port` type consists of a 16-bit number and one of four different protocols: `tcp`, `udp`, `icmp`, and `unknown`. For ICMP, the port number is a composite value of two 8-bit numbers, where the first 8 bits represent the ICMP message type and the remaining 8 bits the message code.

The port index $\mathbb{P} = \langle \Phi^{16}, T \rangle$ consists of an index Φ^{16} for the 16-bit port number and a single-component equality-coded index $T = K^{\langle 4 \rangle}$ for the four different port types. To add a port value $x = \langle x_n, x_t \rangle$ with number x_n and type x_t, we compute:

$$\mathbb{P} \twoheadleftarrow x^{(\alpha)} \;\equiv\; \begin{cases} \Phi^{16} \twoheadleftarrow x_n^{(\alpha)} \\ T \twoheadleftarrow x_t^{(\alpha)} \end{cases} \tag{4.18}$$

Looking up a port value involves computing:

$$\mathbb{P} \circ x \;\equiv\; \begin{cases} \Phi^{16} \circ x_n \wedge T = x_t & \text{if } x_t \neq \texttt{unknown} \\ \Phi^{16} \circ x_n & \text{if } x_t = \texttt{unknown} \end{cases} \tag{4.19}$$

4.2.9 Container Indexes

VAST features the container types `vector`, `set`, and `table`. Like `string`, a container contains a variable number of elements. But unlike `record`, which allows for composing heterogeneous data under named fields, container elements have all the same type, a fixed length, and no field names. In practice containers occur, for example, when describing DNS lookups, where a single host name has multiple A records associated with it. A common query over containers involves subset relationships.

Let us first focus on `vector` and `set`. The difference between the two is that `vector` has an order and may contain duplicates, whereas `set` has no order and cannot include duplicates. Let M be 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 $\mathbb{V}_i$. This structure very much resembles the string index. In fact, we can treat the container index as a generalization of the string index, which operates on values instead of characters as elements. These two indexes only differ in their lookup algorithms. To add a vector value $x = [x_1, \ldots, x_n]$ to a vector index, or a `set` value $x = \{x_1, \ldots, x_n\}$ to a set index, we record the size and each element:

$$\mathbb{X}^V \twoheadleftarrow [x_1, \ldots, x_n]^{(\alpha)} \;\equiv\; \mathbb{X}^S \twoheadleftarrow \{x_1, \ldots, x_n\}^{(\alpha)} \;\equiv\; \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \mathbb{V}_i \twoheadleftarrow x_i^{(\alpha)} & \forall 1 \leq i \leq n \end{cases} \tag{4.20}$$

To perform a subset lookup of a value $x = \{x_1, \ldots, x_n\}$ in the set index $\mathbb{X}^S$, we compute:

$$\mathbb{X}^S \circ x \;\equiv\; \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{M} \mathbb{V}_j = x_i \right) \tag{4.21}$$

This procedure only works for sets where the element order does not matter. Looking up subsets of vectors is isomorphic to substring search, which we describe in Equation 4.10. In fact, we can use subset lookup on strings as well, e.g., to test whether a string contains a certain set of characters, regardless of their order of appearance.

Next, we focus on `table` data. A `table` consists of a sequence of n key-value pairs $\{k_i \rightarrow v_i\}$ for $1 \leq i \leq n$. We consider three query types:

1. Does the table contain the key x?
2. Does the table contain the value y?
3. Does the table contain the mapping $x \rightarrow y$?

To answer these queries, we define the *table index* $\mathbb{X}^T = \langle \phi, X_1, \ldots, X_M, Y_1, \ldots, Y_M \rangle$ as an index $\phi = K^\beta$ for the table size, a sequence of value indexes $X_i = \mathbb{V}$ for the table keys (domain), and a sequence of value indexes $Y_i = \mathbb{V}$ for the table values (codomain). Adding a table value $x = \{x_1 \rightarrow y_1, \ldots, x_n \rightarrow y_n\}$ follows as an extension from set/vector addition:

$$\mathbb{X}^T \twoheadleftarrow \{x_1 \rightarrow y_1, \ldots, x_n \rightarrow y_n\}^{(\alpha)} \;\equiv\; \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ X_i \twoheadleftarrow x_i^{(\alpha)} & \forall 1 \leq i \leq n \\ Y_i \twoheadleftarrow y_i^{(\alpha)} & \forall 1 \leq i \leq n \end{cases} \tag{4.22}$$

To perform a table key lookup of a value x, we compute:

$$\mathbb{X}^T \circ x \;\equiv\; \begin{cases} \bigvee_{i=1}^{M} X_i = x & \text{if } \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni x} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.23}$$

To perform a table value lookup of a value y, we compute:

$$\mathbb{X}^T \circ y \;\equiv\; \begin{cases} \bigvee_{i=1}^{M} Y_i = y & \text{if } \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni y} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.24}$$

To perform a lookup for a mapping $x \to y$, we compute:

$$\mathbb{X}^T \circ y \;\equiv\; \begin{cases} \bigvee_{i=1}^{M} (X_i = x \wedge Y_i = y) & \text{if } \circ \in \{\in\} \\ \overline{\mathbb{X}^T \ni (x \to y)} & \text{if } \circ \in \{\notin\} \end{cases} \tag{4.25}$$

4.3 Query Processing

The first step towards efficient query execution consist of rewriting the expression in a canonical form. We describe this process in §4.3.1. After having normalized the expression and optimized it for efficient execution, VAST's fully asynchronous index engine can work on the individual expression components in parallel. §4.3.2 details this procedure. When it comes to extracting actual index hits from the archive, the export component coordinates the communication between INDEX and ARCHIVE, which involves a significant amount of pipelined communication. We show how finite state machines help with implementing this communication effectively in §4.3.3.

4.3.1 Expression Normalization

When users issue a query, they can express the same semantics in various syntactic forms. For example, the expressions $\overline{A \wedge B}$ and $\overline{A} \vee \overline{B}$ differ syntactically but not semantically. Likewise, a user may equivalently express a predicate as $I = x$ or $x = I$, because the equality relation is symmetric. To facilitate processing, query engines *normalizes* the expression AST into a canonical form. Normalization not only reduces equivalence classes of queries to a single point, but also enables optimizations to speed up execution, as we describe in §4.3.2.

VAST currently performs three transformations of the AST:

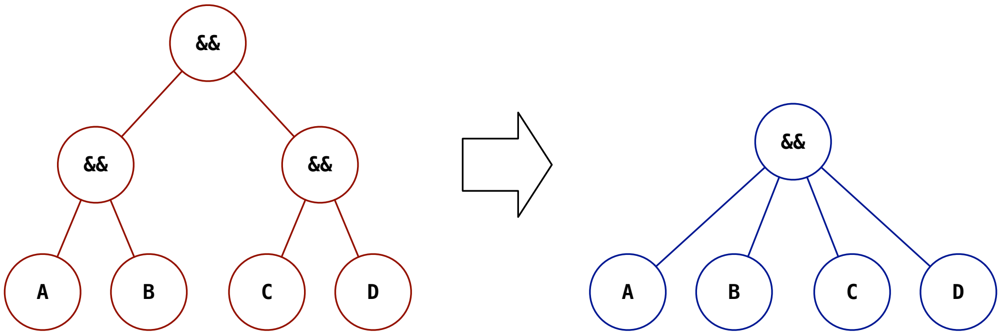


Figure 4.6: Expression normalization: *hoisting*. To reduce the number of intermediary AST nodes, the normalization hoists nested conjunctions and disjunctions. This figure illustrates hoisting conjunctions: $(A \wedge B) \wedge (C \wedge D)$ flattens out to $A \wedge B \wedge C \wedge D$.

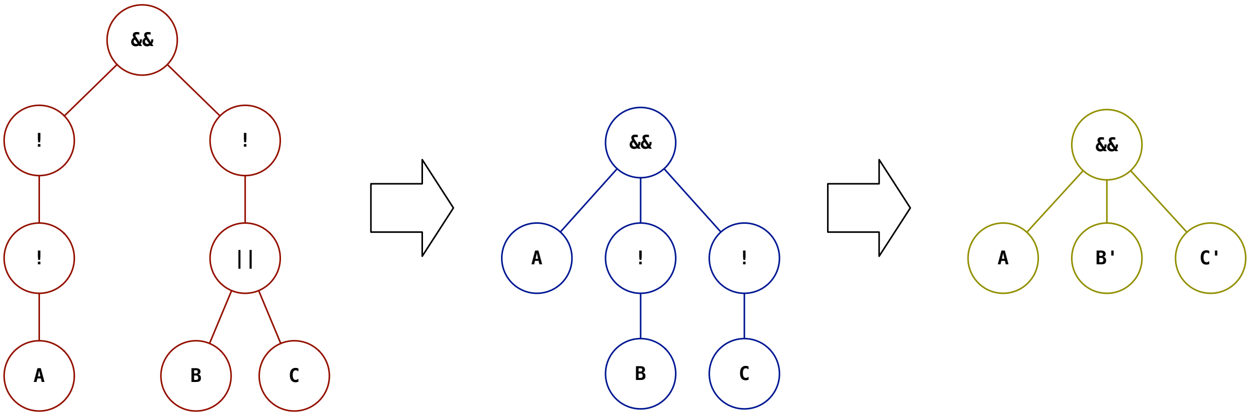


Figure 4.7: Expression normalization: negation normal form (NNF) and negation absorbing. Converting an expression into NNF (i) pushes negations downwards to the predicate level using De-Morgan and (ii) eliminates double negations. The expression $\overline{\overline{A}} \wedge \overline{B \vee C}$ first becomes $A \wedge \overline{B} \wedge \overline{C}$. Thereafter, we can absorb negations further and reduce the intermediate expression to $A \wedge B' \wedge C'$. (For example, if $B = \overline{I < x}$, then $B' = I \geq x$.)

Hoisting. The normalization process hoists nested conjunctions and disjunction to reduce the height of the AST and avoid unnecessary evaluations. Figure 4.6 exemplifies this process with conjunctions; it equally applies to disjunctions.

Aligning. A predicate of the form $LHS \circ RHS$ can have two forms when either side contains a value. The alignment step ensures that the extractor always occurs on the LHS and the value on the RHS. For example, the normalization procedure rewrites the expression $42 > I$ to $I < 42$, flipping LHS and RHS and also the relational operator.

NNF. Normalization also brings the AST in *negation normal form* (NNF). This step includes two aspects: (i) pushing negation inwards to the predicate level and (ii) eliminating double negations. For example, the NNF of $\overline{\overline{A}} \wedge \overline{B \vee C}$ is $A \wedge \overline{B} \wedge \overline{C}$. The first half of Figure 4.7 illustrates this example.

Absorbing. Since each relational operator has a complement, we can remove negations entirely. For example, $\overline{I < x}$ becomes $I \geq x$. The second half of Figure 4.7 illustrates this example. After converting the AST to NNF and absorbing remaining negations, we perform another hoisting pass to ensure that the changed AST does not include newly created inefficiencies.

4.3.2 Evaluating Expressions

When the index receives a query consisting of multiple predicates combined with boolean operators, efficient evaluation becomes a vital concern in order to deliver interactive response times. In the workflow for historical queries, which we describes in §3.2.4.1, 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 1-bits), PARTITION forwards them to INDEX, which in turn relays them to EXPORTER.

To minimize latency and relay hits upstream as soon as possible, we normalize queries to negation normal form (NNF) and absorb remaining negations (see §4.3.1). The absence of negations, aside from saving an extra complement operation, has a useful property: a 1-bit will never turn to 0 when evaluating disjunctions. 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 `:addr` 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 newly arriving hits 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 a false positive, but without negations, we can relay this change immediately since a 1-bit cannot turn 0 again in a disjunction. In other words, the absence of negations makes it possible to relay hits as soon as they manifest.

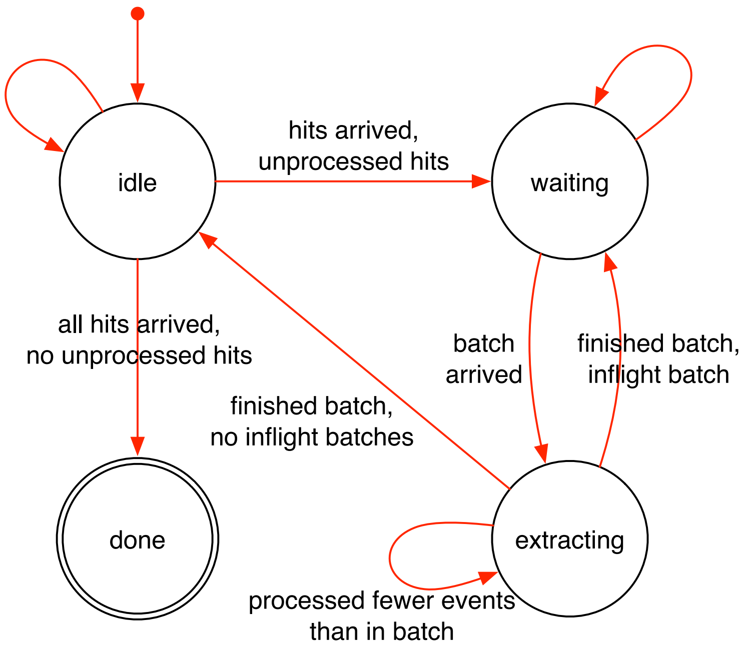


Figure 4.8: The QUERY state machine.

4.3.3 Finite State Machines

We found that finite state machines (FSMs) provide an indispensable mechanism to ensure correct message handling during query execution. Recall from §3.2.4 that NODE spawns an EXPORTER for each query to bridge ARCHIVE and INDEX. We implemented EXPORTER as a finite state machine (see Figure 4.8), which begins in state `idle`. Upon receiving new hits, EXPORTER asks ARCHIVE for the corresponding batches and transitions to `waiting`. As soon as the first batch arrives, it transitions to `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.

4.4 Code Base

As of this writing, VAST comprises 32,300 lines of C++ code, excluding whitespace and comments, plus 6,700 lines of unit tests verifying the system's building blocks and basic interactions. We distribute VAST as free open-source software at `http://vast.io` under a permissive 3-clause BSD license.

VAST uses CMake [48] for cross-platform builds. Aside from CMake, VAST only depends on the C++ Actor Framework (CAF) [33]. In principle, CAF (and therefore VAST) can run on any POSIX-compliant UNIX platform. We tested VAST primarily on recent Linux and FreeBSD distributions. CAF has no dependencies other than a standard-compliant C++11 compiler. VAST, however, requires a C++14 compiler, because we make use of features not available in C++11. These include improved function return type deduction, variable templates, aggregate member initialization, generic lambdas, lambda capture expression, as well as several standard library improvements including user-defined literals and additional type aliases for template metaprogramming.

Type	Structure	Append	Lookup
basic*	I	$I \twoheadleftarrow x^{(\alpha)}$	$I \circ x$
k-component†	$K^\beta = \langle I_k, \dots, I_1 \rangle$ $\Theta^k = K^\beta \quad {\scriptstyle \beta_i = \beta_j = 2 \,\wedge\, \lvert\beta\rvert = k}$ $\Phi^w = K^\beta \quad {\scriptstyle \prod_{i=1}^{k} \beta_i \leq 2^w}$	$K^\beta \twoheadleftarrow x^{(\alpha)} \;\equiv\; I_i \twoheadleftarrow x_i^{(\alpha)} \quad \forall 1 \leq i \leq k$	$K^\beta \circ x \;\equiv\; \ell(K^\beta, \circ, x)$
`bool`	$\mathbb{B} = S$	$\mathbb{B} \twoheadleftarrow x^{(\alpha)} \;\equiv\; S \twoheadleftarrow \alpha \quad \text{iff } x = \texttt{true}$	$\mathbb{B} \circ x \;\equiv\; \begin{cases} \overline{S} & x = \texttt{false} \\ S & x = \texttt{true} \end{cases}$
`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`	$\mathbb{S} = \langle \phi, \kappa_1, \dots, \kappa_M \rangle$ $\phi = K^\beta$ $\kappa_i = \Theta^8$	$\mathbb{S} \twoheadleftarrow \langle x_1, \dots, x_n \rangle^{(\alpha)} \;\equiv\; \begin{cases} \phi \twoheadleftarrow n^{(\alpha)} \\ \kappa_i \twoheadleftarrow x_i^{(\alpha)} \quad {\scriptstyle \forall 1 \leq i \leq n} \end{cases}$	$\mathbb{S} \circ x \;\equiv\; \begin{cases} \mathbb{0} & \lvert x\rvert > M \\ \phi = 0 & \lvert x\rvert = 0 \\ \phi = \lvert x\rvert \wedge \bigwedge_{i=1}^{\lvert x\rvert} \kappa_i = x_i & \circ \in \{=, \neq\} \\ \phi \geq \lvert x\rvert \wedge \bigvee_{i=1}^{M - \lvert x\rvert + 1} \left(\bigwedge_{j=1}^{\lvert x\rvert} \kappa_{i+j-1} = x_j \right) & \circ \in \{\in, \notin\} \end{cases}$

* The *basic* index has a fixed binning, coding, and compression scheme and operates on values $x \in X \subseteq \mathbb{N}_0^+$. It has cardinality $C \leq |X|$. (see §2.4.3)

† The k-component index operates with a base $\beta = \langle \beta_k, \dots, \beta_1 \rangle$. We introduce algorithm ℓ in §2.4.4. The bit-sliced index [199] is a special case of K^β where $\beta_i = 2$ for all $i \in \{1, \dots, k\}$. 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 4.1: Summary of append and lookup operations on high-level indexes.

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

* 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 4.2: Summary of append and lookup operations on high-level indexes.

Chapter 5

Evaluation

Le véritable voyage de découverte ne consiste pas à chercher de nouveaux paysages, mais à avoir de nouveaux yeux.

Marcel Proust

After presenting the design and implementation of VAST in previous chapters, we now evaluate our system across several dimensions. We begin with introducing our measurement infrastructure and data sets in §5.1. We continue with a qualitative analysis of correctness in §5.2. Thereafter, we perform a quantitative analysis of ingestion and indexing throughput (§5.3), query latency (§5.4), intra-machine and inter-machine scaling (§5.5), and storage overhead (§5.6).

5.1 Measurement Infrastructure

We conduct our measurements on multiple machines using several data sets. Before discussing the specifics of the analysis, we describe our infrastructure, which includes our testbeds in §5.1.1 and data in §5.1.2.

5.1.1 Machines

Our available machines for testing consist of individual machines as well as a cluster. We conduct our single-machine measurements on PACKRAT: this machine comes with two 8-core Intel Xeon E5-2640 CPUs clocked at 2 GHz, 128 GB RAM, six 4 TB 6 G SAS 7.2 K disks configured in a RAID 10, and a RAID controller with 2 GB cache. PACKRAT runs 64-bit FreeBSD 10.2-RELEASE and stores its data on a ZFS filesystem.

For our distributed measurements, we employ a cluster with 28 BLADE worker nodes and a MANAGER node, connected via 1 GE on the same switch. Each BLADE comes with 12 GB

Type	Events	Size	Dimensionality
Test	10 M	N/A	1*
PCAP	10 M	6.4 GB	5†
Bro	3.4 M	411 MB	20‡
Bro-N	1.24 B	152 GB	20‡
PCAP	877 K	850 MB	5†
Bro	28 K	5.3 MB	1–27¶

* We used an IP address value whose bytes follow a $Pareto(0, 1)$ distribution.
† We only index the connection 5-tuple and skip the packet payloads.
‡ We perform the measurements with Bro's 20-column `conn.log`.
¶ We concatenate all logs that Bro produced by processing the trace.

Table 5.1: Data sets used for our evaluation. The top section of the table describes the data we use for our single-machine experiments, the middle section for our cluster measurements, and the bottom section for our compression algorithm comparison.

RAM and two 8-core Intel Xeon E5430 CPUs clocked at 2.66 GHz. The MANAGER hosts a 14 TB RAID-6, shared among the BLADEs via NFS. Each BLADE additionally holds 1 TB locally as a ZFS RAID-1 of two SATA disks. The BLADEs run FreeBSD-10.2-CURRENT and the MANAGER runs FreeBSD-10.3-RELEASE.

5.1.2 Data Sets

Our measurements consist of two types of input: 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, which enables specification of dimensionality and value distribution. This SOURCE produces all synthetic data in memory to avoid adding I/O load. For real-world input, we use PCAP traces plus the derived Bro logs. For plain PCAP ingestion, VAST functions as a flow-oriented bulk packet recorder akin to Time Machine [130]. We summarize key properties of our data sets in Table 5.1.

Our PCAP trace for single-machine evaluation represents live traffic recorded at the uplink of the International Computer Science Institute (ICSI) at Berkeley, California. The Bro connection logs for our cluster evaluation stem from network traffic recorded over the course of 3 days in fall 2015 at a large university campus. For our compression algorithm comparison, we use a synthetically generated PCAP trace from the M57-Patents scenario [128].

Moreover, we use the set of test queries given in Table 5.2, which a security operator for a large enterprise confirmed indeed reflect common searches during an investigation. The top 6 queries execute over Bro connection logs and the bottom 3 over PCAP traces.

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 5.2: Test queries for throughput and latency evaluation. The top 6 queries run over Bro connection logs and the bottom 3 over a PCAP trace.

5.2 Correctness

Per §4.4, VAST comes with 6,700 lines of unit tests checking the system's building blocks. We expand on these checks with an end-to-end test of whether the entire pipeline—from import to querying to export—yields correct results. For validation, we processed our ground truth (PCAP traces and Bro logs) separately and cross-checked against the query results VAST delivers. We found full agreement.

5.3 Throughput

One key performance metric is the rate of events that VAST can ingest. Recall from §3.2 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 data, we did not optimize SOURCEs at this stage in our development, nor ARCHIVE, which merely sequentially compresses events into fixed-size chunks and writes them out to the filesystem. Instead, we concern ourselves with achieving high performance at the bottleneck: INDEX, which performs the CPU-intensive task of building bitmap indexes.

Per §3.2, INDEX uses PARTITIONs as its unit of horizontal data scaling. Initially, we load-balanced each arriving event batch over *multiple* PARTITIONs, but later found that varying the number of PARTITIONs has no effect on performance. This stems from the fact that each PARTITION already exhibits a high degree of concurrency by spawning numerous INDEXERs, one per batch. This abundance of tasks fully saturates the worker queues of CAF's actor scheduler. Consequently, we perform all ingestion measurements with only one active PARTITION and replace it with a new one once it reaches its default capacity of 1 M events.

We measure throughput at various points during the import process. The setup consists of a single VAST process which includes all components. This presents the worst-case scenario, because now a single process has to perform both CPU-intensive tasks and I/O at the same time. Normally SOURCEs run as separate process at the data producer, which alleviates I/O load and CPU-intensive data parsing, at the cost of extra message serialization between SOURCE and IMPORTER.

In particular, we look at the number of events per second that ARCHIVE, INDEX, and SOURCE can sustain when varying the input batch size as well as the number of CPU cores we allow the runtime to use. We control the number of cores by adjusting the threads of the scheduler in CAF. The data formats in our measurement include Bro, PCAP, and our benchmark test (see Table 5.1). The highly concurrent architecture complicates measurements of aggregate throughput at INDEX, which runs multiple INDEXERs in parallel, but not all start and finish at the same time. The overlapping and shifted execution times prevent an accurate measurement of an aggregate rate at a given point in time. Therefore we compute INDEX throughput as the number of events processed between start and end of the *entire* measurement. Thus, the

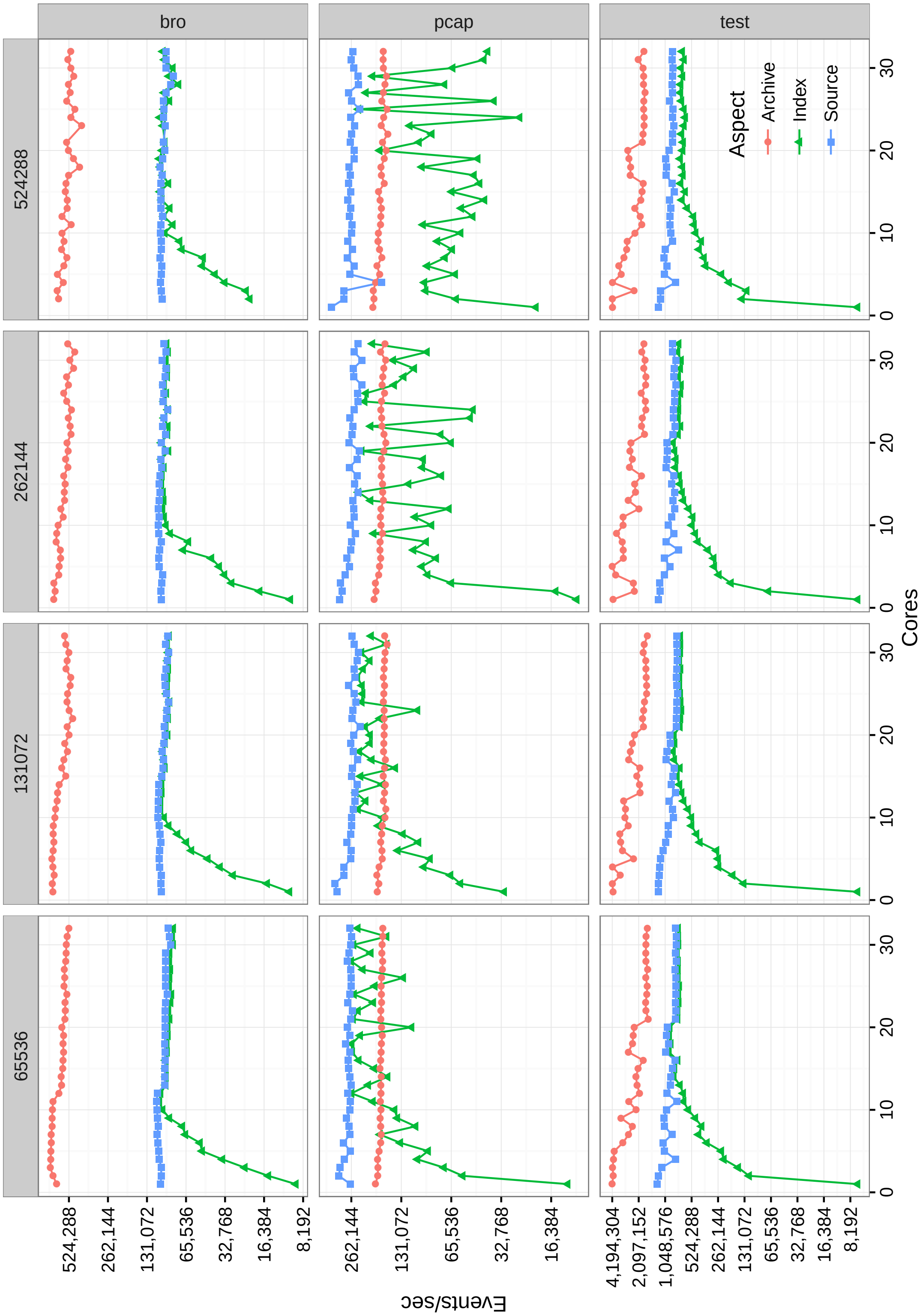


Figure 5.1: Throughput measured at various points during import. Each panel shows a combination of data format and batch size.

throughput can never exceed the input rate of SOURCE, which dictates the upper bound for INDEX.

Figure 5.1 summarizes the results. Each panel shows a combination of data format and batch size. The y-axis shows throughput in events per second; note the log scale. As mentioned above, by the current design, SOURCE and ARCHIVE exhibit fairly constant throughput. We experimented with four different batch sizes at SOURCE: 65,536, 131,072, 262,144, and 524,288. These values represent the number of events SOURCE buffers before sending them to IMPORTER. Increasing the batch size has a particularly pronounced effect for PCAP events. For this format, events consist of the connection 5-tuple plus the full IP payload. The packets in the trace have a median wire length of 1434 bytes. That is, these events exhibit a size that is one order of magnitude higher than Bro events, and another order of magnitude higher than the test events. With a batch size of 524,288, we see deteriorating throughput because of large message sizes: a batch can occupy as much as 746 MB, assuming all packets have the maximum MTU of 1492 bytes. To avoid the higher latency jitter induced by large batch sizes, we restrict the following measurements to 65,536 events per batch.

We observe that the indexing rate approaches the input rate for all sources between 8–16 cores. More cores yield no further improvement; in fact, performance decreases slightly. This artifact likely stems from over-subscription: our measurement system PACKRAT has hyperthreading enabled, which exposes 32 cores to the OS instead of the 16 phyiscal ones. Moreover, CAF does not pin its worker threads to a specific core. This means that the OS has the freedom to schedule the same thread on a different core. CAF comes with a work-stealing scheduler which increases the probability that the same actor executes on a different thread. These effects can increase the number of context switches and cache evictions, resulting in poorer performance. The CAF developers are aware of this situation and further work on a NUMA-aware scheduler.

We take a closer look at the indexing throughput of Bro events for a batch size of 65,536 in Figure 5.2. Each panel shows one run with a different number of cores. Each row represents a single batch, where the vertical bar denotes the time when indexing began and the circle when it completed. The "tail" corresponds to the time it took to index an event batch. Starting at 11 cores, we see a uniform processing time of event batches, which aligns with the results mentioned in the previous paragraph.

Looking at the measurements with a batch size of 65,536, 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 5-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

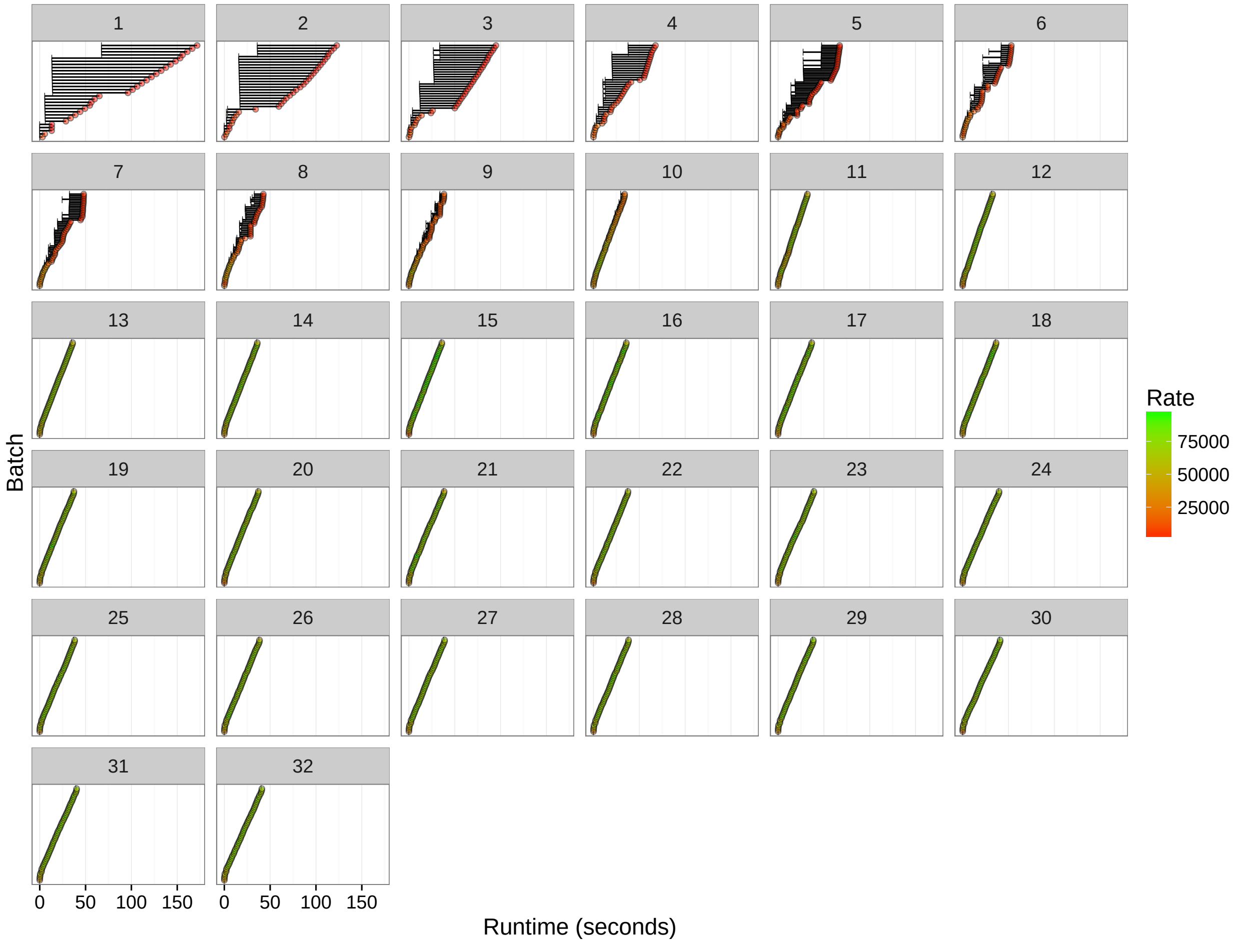
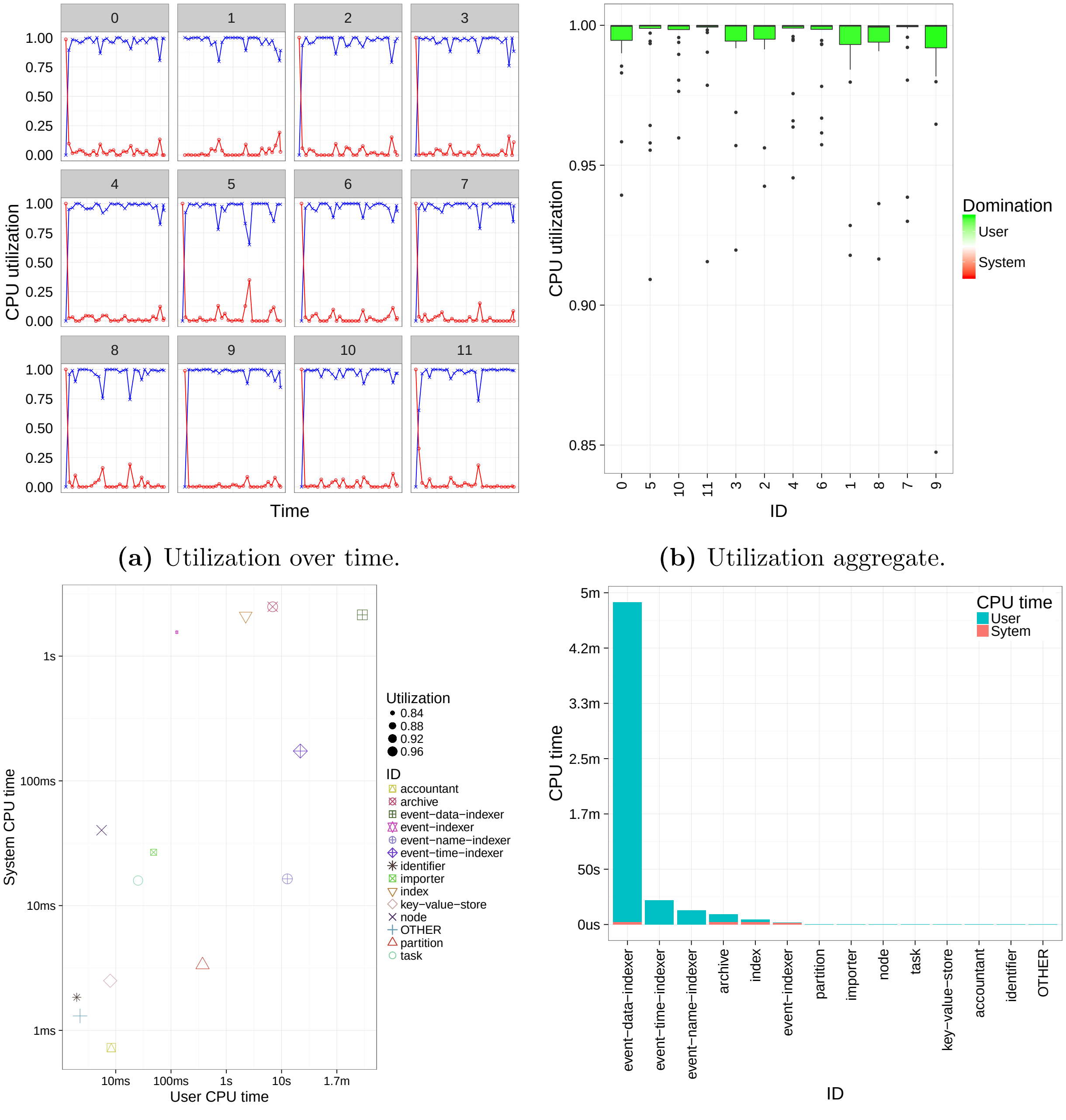


Figure 5.2: Indexing runtime for event batches of size 65,536. Each panel shows one run with a different number of cores. Each row represents a single batch, where the vertical bar denotes the time when indexing began and the circle when it completed.

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.

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` on FreeBSD/Linux and `thread_info` on Mac OS), i.e., we only measure actor execution and leave CAF runtime overhead out of the picture. We contributed our profiling enhancements to the CAF code base, including the R scripts to automatically generate over a dozen similar plot variations.

We begin with looking at the CPU utilization of CAF's worker threads in Figure 5.3. The left panel Figure 5.3(a) shows CPU utilization as function of time, where the bottom (red) line represents system CPU time and the top (blue) line user CPU time. The plot exhibits



(a) Utilization over time.

(b) Utilization aggregate.

(c) Utilization per actor type.

(d) Utilization per actor type.

Figure 5.3: CPU utilization when indexing Bro event batches of 65,536 with 12 cores. The top two Figures (a) and (b) show CPU utilization per worker thread, while the bottom two Figures (c) and (d) show actor-level utilization. Figure 5.4 decomposes Figure (c) to the level of actor instances.

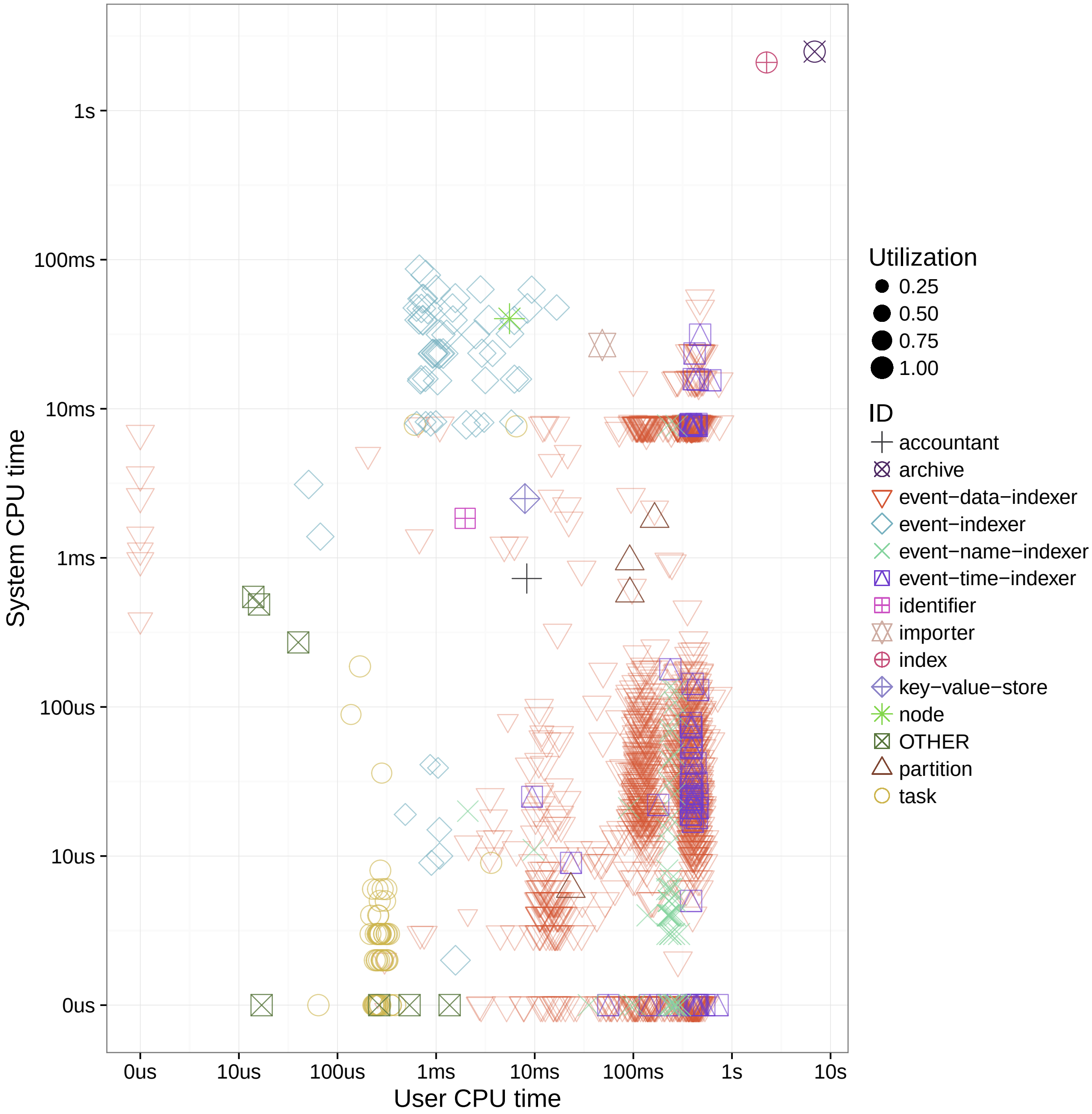


Figure 5.4: CPU utilization per actor instance during import. The x-axis shows user time and the y-axis system time; note the log-scaling. Each point represents a single instance of an actor. Figure 5.3(c) shows an aggregate version of this plot.

one panel per worker thread. Overall, we see full CPU utilization of all threads in user time, with a few occasional spikes when VAST writes out indexes to the filesystem. Even though VAST performs blocking I/O writes, the operation does not incur significant CPU stalls, presumably because the OS provides a large enough buffer cache and writes out the data asynchronously at its convenience. The right panel Figure 5.3(b) aggregates utilization into one boxplot per thread. Except for a few outliers, the median utilization approaches 1.

Next, we seek to better understand the performance of the individual components in terms of actors. In Figure 5.4, 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, which we define as 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 primarily manages PARTITIONS and builds "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, which we can see accumulating on the right-hand side, because building bitmap indexes is a CPU-bound task. Figure 5.3(d) summarizes the individual actor instances per actor type. Each bar represents the aggregate amount of CPU time of a specific actor type. Overall, we see that EVENT-DATA-INDEXER consumes the largest amount of CPU time, followed by EVENT-TIME-INDEXER and EVENT-NAME-INDEXER, which record event meta data. This observation agrees with our expectation: constructing indexes over the data consumes most time. Moreover, this task does not involve I/O other than writing out the indexes to the filesystem, which explains the dominance of user CPU time.

Overall, we find that VAST makes near optimal use of available CPU resources (in terms of no idling) when indexing events: all worker threads in CAF's scheduler exhibit full CPU utilization, with the actors responsible for indexing accounting for the largest share.

5.4 Latency

Query response time plays a crucial role in assessing the system's viability. The iterative workflow of investigations requires an interactive experience, which demands latencies on the order of seconds. To measure latency, we use the set of test queries given in Table 5.2, which a security operator for a large enterprise confirmed indeed reflect common searches during an investigation.

There exist several latency aspects across the entire query pipeline. For analysts, a particularly important metric represents the time until the first "taste" arrives, i.e., the time it takes from issuing a query until the first results show up on the screen. The taste constitutes an important metric because small data subsets already allow for quick triaging decisions to steer the analysis process, thereby avoiding unproductive analyst downtime. From an internal system perspective, there exist additional aspects of latency. VAST spawns one EXPORTER per query, which acts as a middleman receiving hits from INDEX and retrieving

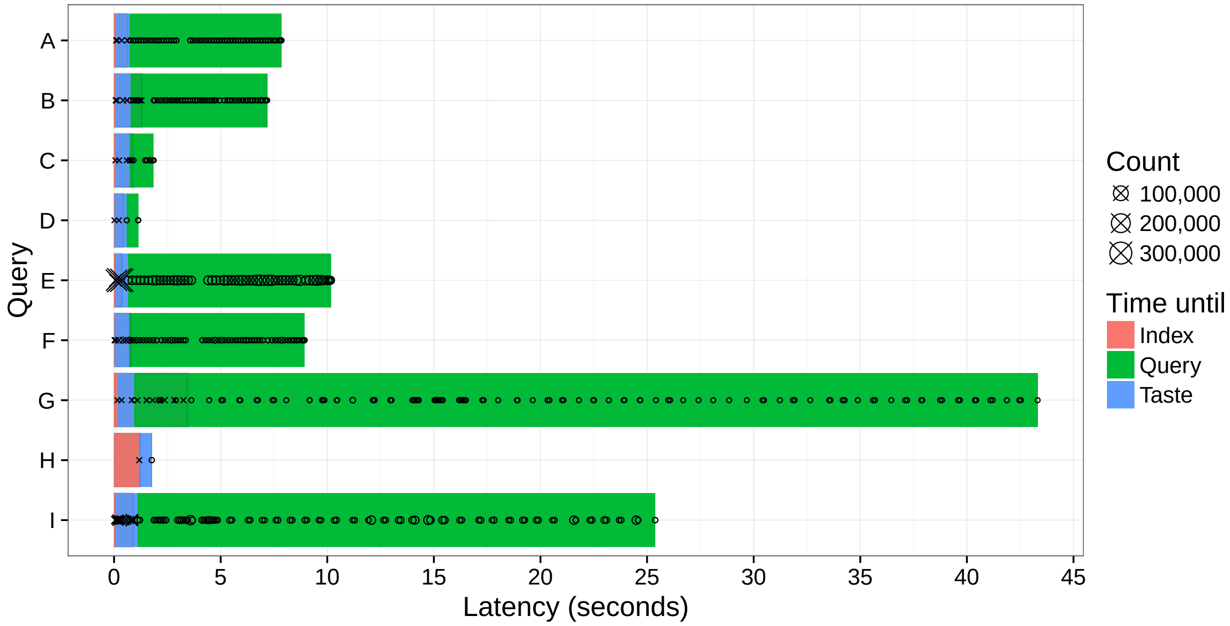


Figure 5.5: Query pipeline reflecting various stages of single-node execution. The first stage (Index) may appear absent because it takes too little time to manifest in the plot. Crosses show when hits arrive, and circles the completion of extracting results from an event batch. The median taste (arrival of first result) is about 1 second.

the corresponding compressed chunks of events from ARCHIVE. Two interleaving latency elements concern 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 successful candidate checks. We take a closer look at these latency aspects in the following.

Figure 5.5 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 blue bar shows the time until EXPORTER has sent the first result to its SINKs. This corresponds to the "taste" time, since from the user perspective it represents the first system response. The green bar shows the time until EXPORTER has sent the full set of results to its SINK. The black transparent 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 (green 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, enabling concurrent sweeps over the candidates. Alternatively, we could offload more computation

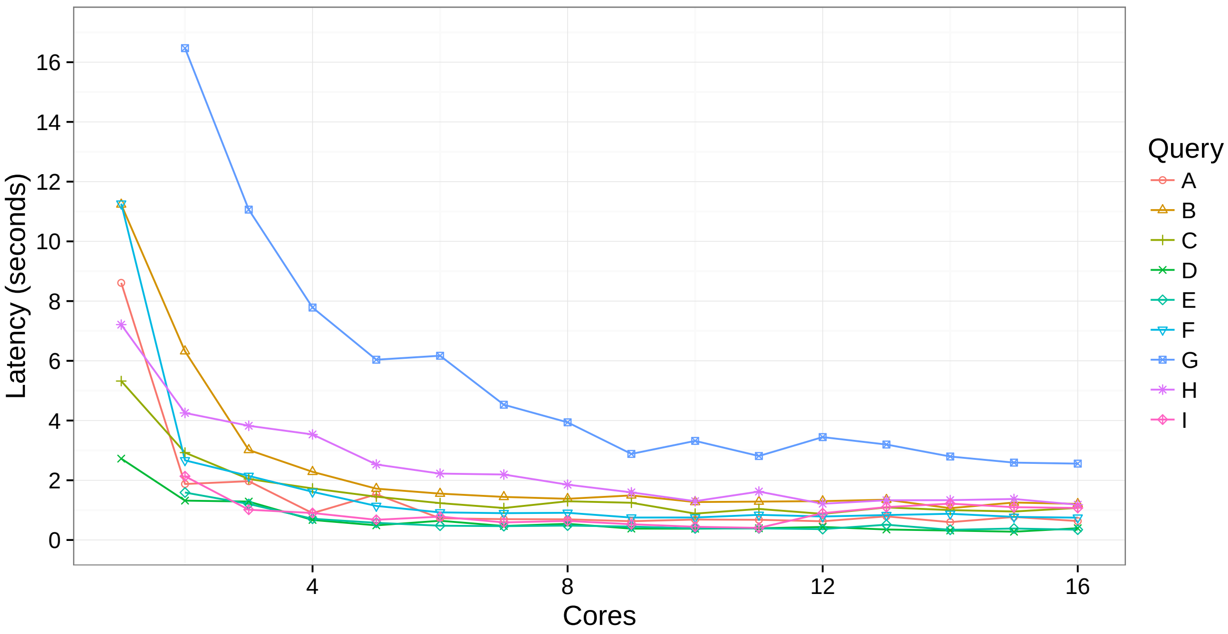
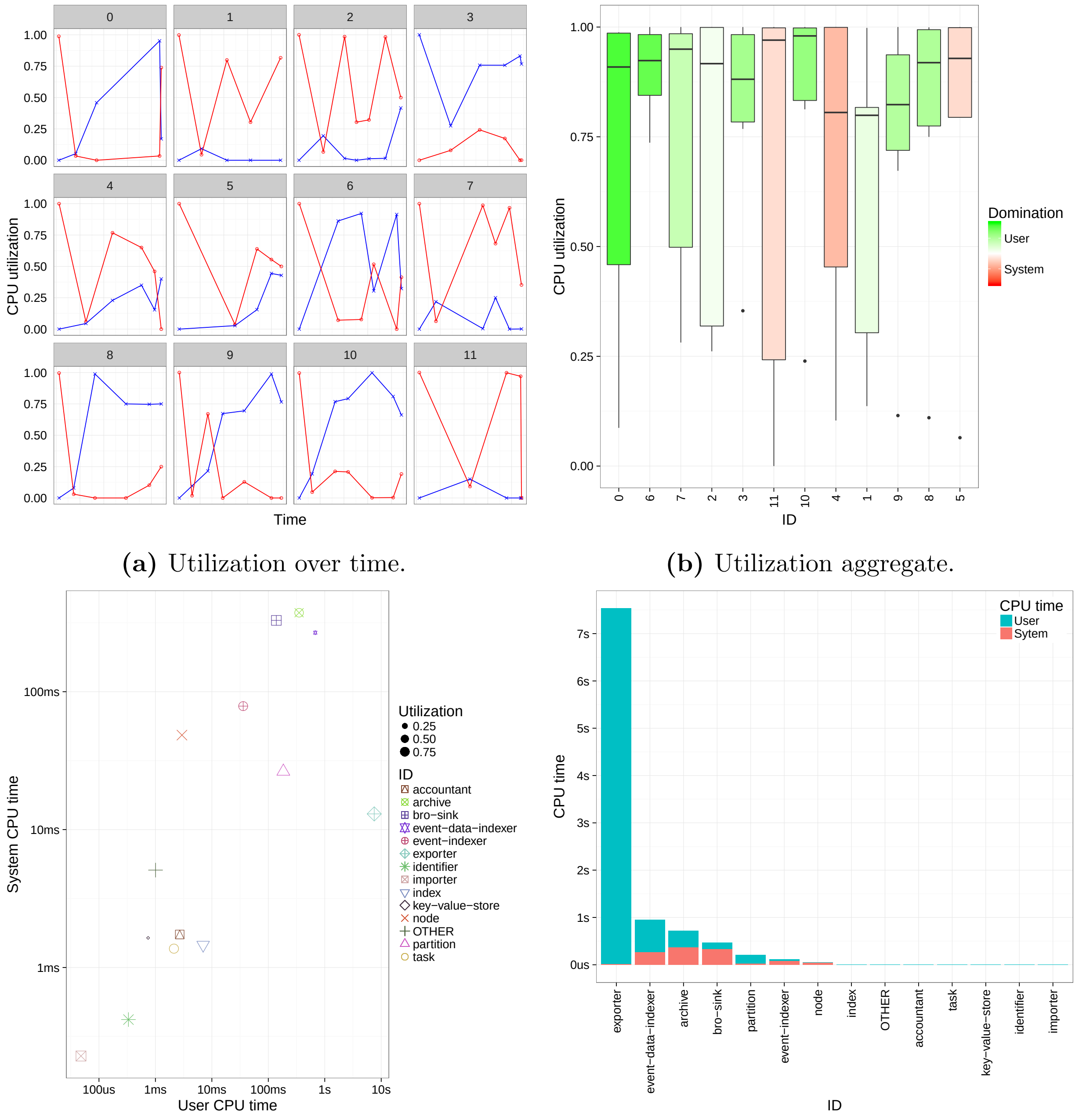


Figure 5.6: Index latency (full computation of hits) as a function of cores.

into ARCHIVE. Selective decompression algorithms [78] present an orthogonal avenue for further improvement. Such algorithms enable decompressing and deserializing events at a finer granularity, which can save a significant number of CPU cycles.

VAST processes index lookups in a continuous fashion, with first hits trickling in after a few 100 msecs. Figure 5.6 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 (`:addr in 192.150.186.0/23 && :port == 3389/?`) processes large intermediate bit vectors during the evaluation, which require more time to combine.

Next, we take a closer look at micro-benchmarks during query processing. Figure 5.7 illustrates the resource utilization during executing query `E`. The semantics of the different figures follow Figure 5.3. Unlike data import, the query export exhibits a more convoluted picture. While ingestion was CPU-bound, we find the query processing to be I/O-bound. Per Figure 5.7(a), we observe significantly more system CPU time (red lines) domination. Because VAST currently performs blocking I/O, we also see poor overall utilization: the two points for CPU and system time do not add up to 1.00. Our most likely explanation for this artifact is that the missing share goes into seek time. Indeed, VAST creates numerous smaller index files, which exacerbate this behavior. Figure 5.7(b) illustrates the poorer utilization. We observe a larger fraction of system CPU time, which lowers the overall utilization in some cases due to extra seeks. Also Figure 5.8 confirms our observation: in comparison to Figure 5.4, the center of the point cloud shifts from the bottom right to the top left, in particular for the EVENT-DATA-INDEXER, which performs the index operations. Still, when looking at overall



(a) Utilization over time.

(b) Utilization aggregate.

(c) Utilization per actor instance.

(d) Utilization per actor type.

Figure 5.7: CPU utilization during execution of query E for event batches of size 65,536 with 12 cores. The top two Figures (a) and (b) show CPU utilization per worker thread, while the bottom two Figures (c) and (d) show actor-level utilization. Figure 5.8 decomposes Figure (c) to the level of actor instances.

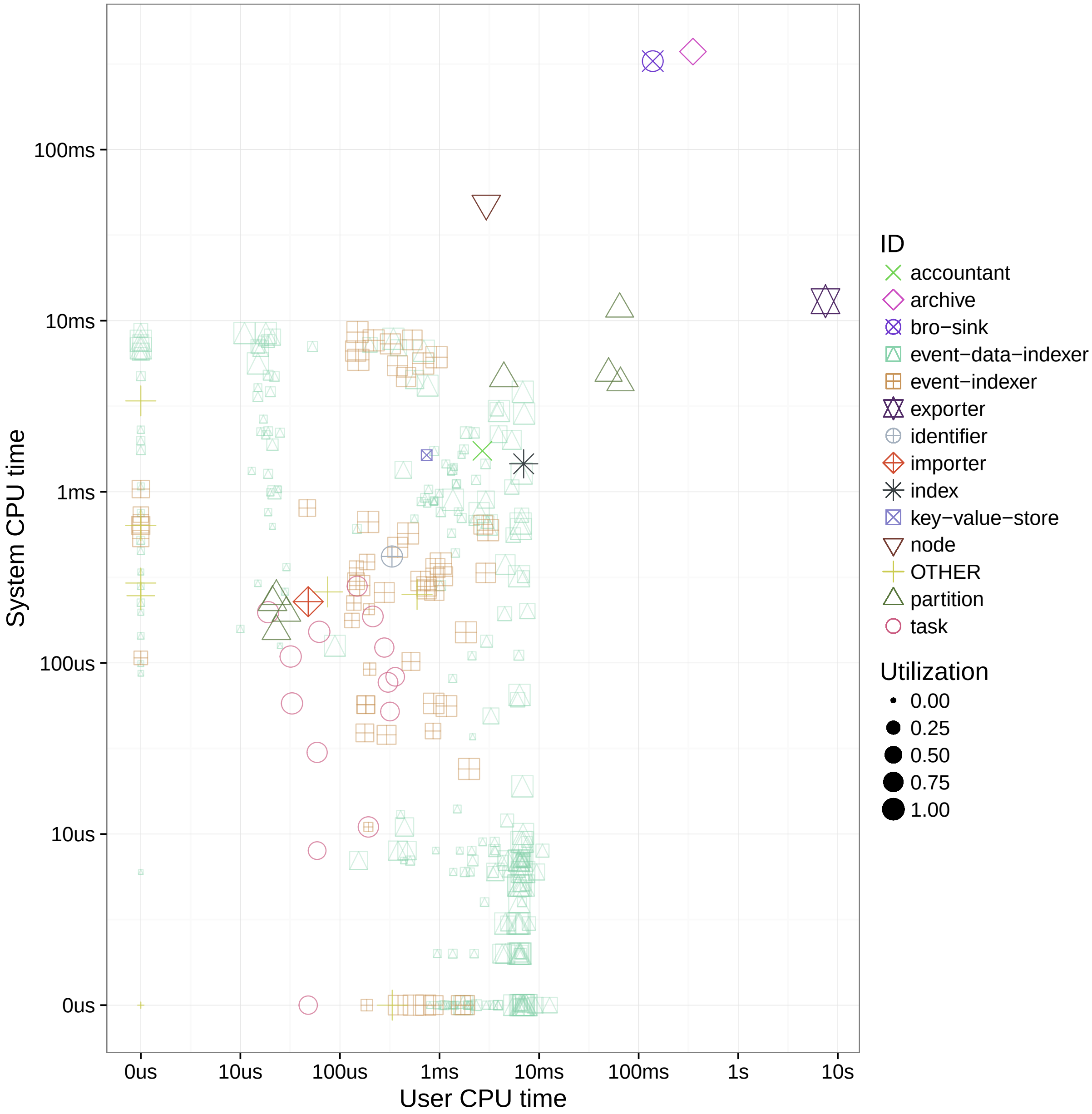


Figure 5.8: CPU utilization per actor instance during export. The x-axis shows user time and the y-axis system time; note the log-scaling. Each point represents a single instance of an actor. Figure 5.7(c) shows an aggregate version of this plot.

CPU time in Figure 5.7(d), we see that EXPORTER accounts for the largest share, because it extracts results from the compressed data blocks received from ARCHIVE and (currently unconditionally) performs a candidate check. However, query E does not require a candidate check and we should add additional logic to prevent the check when the index result is exact. Other research has also found that decompressing the base data and performing the candidate check dominates query execution [78, 162], which aligns with our findings.

Overall, we find that VAST meets our single-machine performance expectations. In particular, we prioritized abstraction to performance in our prototype 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 further gains.

5.5 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 with copious amounts of data.

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 (see §5.1) by load-balancing them over the cluster NODEs in batches of 65 K. That is, 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 5.9 shows per-machine CPU inverse utilization $1/U$ for

$$U = \frac{\sum_i^N (u_i + s_i)}{\sum_i^N t_i}$$

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.

Our second measurement seeks to understand how query latency changes when varying the number of nodes. We show the index completion latency of query D in Figure 5.10. 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). We observe linear scaling from 12 nodes upward, but experience problems for the lower half.

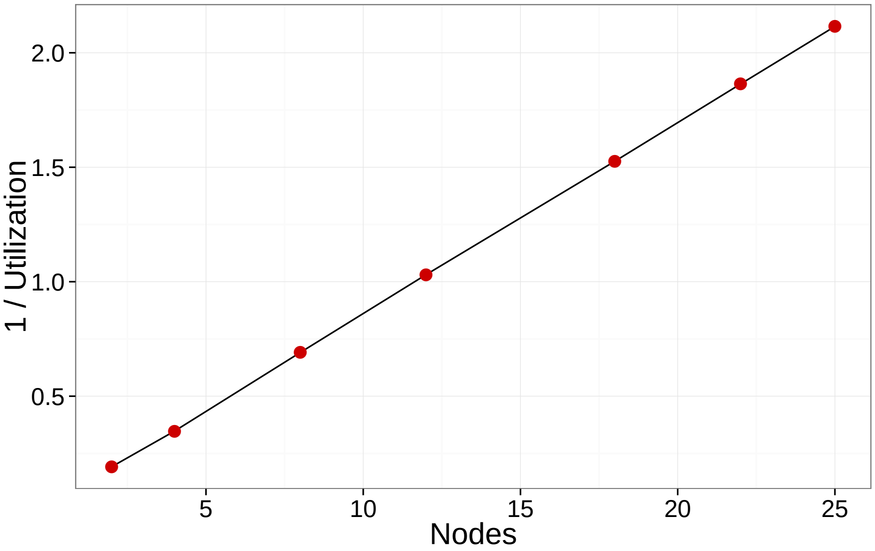


Figure 5.9: Per-node CPU utilization during ingestion. A utilization of 1 does not mean full saturation, because each process runs multiple threads. For example, for 2 nodes we observe a per-node utilization of $U = 5.2$, which we here as $1/U = 0.2$.

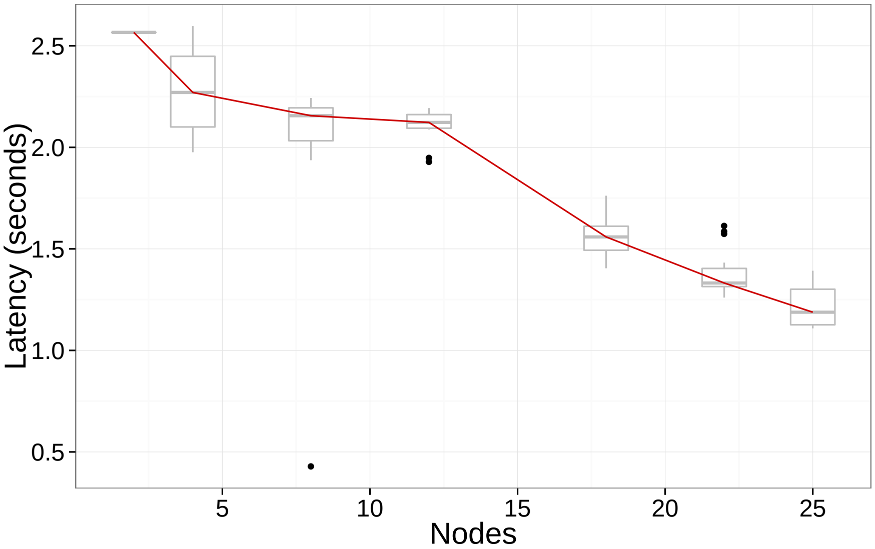


Figure 5.10: Index completion latency as function of nodes for query D. The line shows the median latency of all involved nodes.

Other queries show linear scaling for small numbers of nodes. We are in the process of investigating the discrepancy.

5.6 Storage

In addition to evaluating VAST during runtime in terms of CPU consumption, we also measure the overhead of VAST's persistent data structure residing on the filesystem. In §5.6.1, we compare various general-purpose compression algorithms, which the archive relies upon to compactly store the base data. In §5.6.2, we examine the size of both archive and index and for our single-machine measurements.

5.6.1 Archive Compression

VAST stores a full copy of the events in the archive, in compressed form (see §3.2.2). The choice of compression determines three central system aspects:

Throughput. The higher the compression speed, the faster VAST can ingest events.

Latency. The higher the decompression speed, the faster VAST can return results.

Storage. The higher the space savings, the more events VAST can store on the filesystem. Moreover, smaller files produce fewer I/O operations, which can affect both throughput and latency.

Ideally, we want to maximize all three dimensions, but in practice, there exists a space-time trade-off. To explore this trade-off, we can apply a similar notion of optimality as we do for coding schemes in §2.4.3. Let I denote input data and $\mathcal{A}$ a compression algorithm. Further, let $\textsc{Compression}(\mathcal{A}, I)$ denote the compression speed in MB per second, $\textsc{Decompression}(\mathcal{A}, I)$ the decompression speed, and $\textsc{Savings}(\mathcal{A}, I)$ the reduction in space relative to the uncompressed input, i.e, $1 - \frac{c}{u}$ where c is the compressed and u the uncompressed input size. We only consider algorithms where $c \leq u$. A similar metric to describe space efficiency is the compression ratio, which is the ratio between uncompressed and compressed input size. An algorithm $\mathcal{A}$ is optimal for a fixed input I if there exists no better algorithm $\mathcal{A}'$ such that:

1. $\textsc{Compression}(\mathcal{A}', I) \geq \textsc{Compression}(\mathcal{A}, I)$
2. $\textsc{Decompression}(\mathcal{A}', I) \geq \textsc{Decompression}(\mathcal{A}, I)$
3. $\textsc{Savings}(\mathcal{A}', I) \geq \textsc{Savings}(\mathcal{A}, I)$
4. at least one inequality of (1), (2), or (3) is strict

Multiple optimal algorithms may exist according to this definition. Unlike with coding schemes, we perform an empirical analysis to establish optimality. To this end, we benchmark 19 general-purpose compression algorithms against two types of input, 4 of them with two

	PCAP			**Bro**		
Algorithm	Compression	Decompression	Savings	Compression	Decompression	Savings
ZPAQ [129]	0.19	0.07	**0.66**	0.25	0.03	0.89
MCM [45]	1.08	0.39	**0.66**	1.98	0.22	**0.90**
LZMA25 [127]	2.83	10.30	**0.66**	2.89	10.23	0.87
LZMA20 [127]	3.64	10.26	**0.66**	3.32	9.84	0.84
LZIP [63]	3.23	9.07	**0.66**	3.06	7.63	0.87
BROTLI11 [178]	0.24	48.21	**0.66**	0.21	26.94	0.88
CSC20 [31]	3.59	8.10	0.65	2.60	7.10	0.87
BROTLI9 [178]	5.55	67.59	0.65	7.30	35.86	0.86
ZSTD [54]	8.50	294.62	0.64	14.95	97.08	0.85
TANGELO [31]	0.93	0.33	0.64	1.09	0.13	0.88
ZLING [122]	28.42	28.15	0.62	40.71	22.67	0.85
BSC [89]	4.39	2.04	0.61	6.82	1.45	0.86
ZMOLLY [123]	0.98	0.39	0.60	4.07	0.61	0.87
ZSTDF [54]	217.55	313.09	0.56	245.42	119.87	0.79
BCM [31]	4.07	1.48	0.56	4.19	0.59	0.84
BZIP2:9 [32]	4.96	6.64	0.53	5.39	4.21	0.81
MINIZ [134]	9.71	99.28	0.45	12.91	52.93	0.79
DEFLATE:9 [3]	13.95	110.02	0.45	16.45	54.92	0.80
LZ4 [53]	25.37	**1179.23**	0.44	28.78	378.20	0.75
BZIP2:1 [32]	5.20	8.56	0.44	6.74	5.14	0.80
DEFLATE:1 [3]	36.50	104.45	0.43	71.91	57.85	0.76
SHRINKER [58]	153.48	523.90	0.42	159.46	234.60	0.71
LZ4F [53]	**302.48**	1050.36	0.41	**291.10**	**392.84**	0.69
RAW	8134.38	9022.78	0.00	5402.26	7477.57	0.00

Table 5.3: Benchmark of various compression algorithms, sorted by space savings for PCAP input. Best results exhibit a bold font.

different parameterizations, yielding 23 algorithm instances in total. We first establish the optimal algorithms within each dimension and then discuss the trade-off space.

As for our data sets, we use a 850 MB PCAP trace consisting of 877,469 packets, plus 5.3 MB of derived the ASCII Bro logs [147], per §5.1. The traffic exhibits the following breakdown in terms of number of connections: 50% DNS, 28% HTTP, 18% unknown, 2.6% DHCP, 1.4% SSL, plus a small remainder of SMTP and FTP. Our C++ benchmark tool reads the raw bytes of the PCAP trace in memory, compresses it, and then uncompresses the compressed data into a new memory buffer. We rely on `bundle` [31], an embeddable compression library, for the implementation of the majority of algorithms. Because `bundle` does not support DEFLATE (implemented in `zlib` [3]) and BZIP2 [32], we supplemented our test harness to use the corresponding native APIs for compression/decompression. The RAW algorithm establishes a reference frame for memory bandwidth, as it does not perform any compression but merely copies data between two buffers. We make available our test suite code as open source software, along with the scripts to generate the plots [186].

Table 5.3 summarizes our results, sorted in descending order with respect to space savings of PCAP input. The values show how throughput and savings form two ends of a spectrum. For example, LZ4 and LZ4F dominate in COMPRESSION and DECOMPRESSION, but achieve the smallest ranking in terms of SAVINGS. Conversely, MCM excels in SAVINGS, but ranks close to last in COMPRESSION and DECOMPRESSION. This raises the question where to position oneself in the spectrum of space versus time. Which algorithms perform strictly better than others across multiple dimensions?

For ease of exposition, we seek to answer this question in the three two-dimensional spectra of SAVINGS versus COMPRESSION, SAVINGS versus DECOMPRESSION, and COMPRESSION versus DECOMPRESSION. First, we consider the throughput versus space savings in Figures 5.11–5.14. For each plot, the x-axis shows SAVINGS as percentage and the y-axis throughput for COMPRESSION (Figure 5.11 and Figure 5.12) and DECOMPRESSION (Figure 5.13 and Figure 5.14). To emphasize the optimal algorithms in this two-dimensional space, we superimpose a convex hull. The algorithms on top of the polygon border are optimal, because they strictly dominate the algorithms inside the polygon. From here on, we do not consider the inner algorithms. In Figure 5.11, we see the optimal algorithms LZ4F, ZSTDF, ZLING, ZSTD, the LZMA family, and MCM—from COMPRESSION-optimal to SAVINGS-optimal. Note the discrepancy of BROTLI9 (5.55 MB/sec) and BROTLI11 (0.24 MB/sec). In the bottom row, observe the three steep declines in DECOMPRESSION from LZ4 to ZSTDF, from ZSTD to BROTLI9, and from BROTLI11 to MCM. This tetramodal characteristic naturally exposes four classes along the spectrum: LZ4 on the speed end, MCM on the compactness end, and ZSTD with BROTLI in the middle. Overall, LZ4 clearly offers highest throughput in both, whereas ZSTD offers a good compromise between space savings and throughput. BROTLI has high space savings and good decompression speed, but ranks poorly in terms of compression speed.

We juxtapose the two throughput dimensions COMPRESSION and DECOMPRESSION in Figure 5.15 and Figure 5.16. The $y = x$ diagonal visually separates the algorithms which compress faster (below the diagonal) from those which uncompress faster (above the diagonal). LZ4 and LZ4F dominate as both COMPRESSION-optimal and DECOMPRESSION-optimal algorithms, but also ranks last in SAVINGS. We illustrate the same information in a bar plot in Figure 5.17 and Figure 5.18, with the algorithms on the x-axis sorted with respect to SAVINGS, showing the most space-efficent algorithm on the left.

Based on these measurement results, we must select algorithms in alignment with VAST's requirements, which we outlined in the beginning of §5.6.1. LZ4 offers the highest throughput and therefore represents an apt choice for high-volume scenarios. As second choice, we consider ZSTD with an order of magnitude lower throughput, but up to 23% higher space savings. From here on, we see another order of magnitude difference in throughput while increasing the savings only by a few percent. Therefore, we only support LZ4 and ZSTD in VAST.

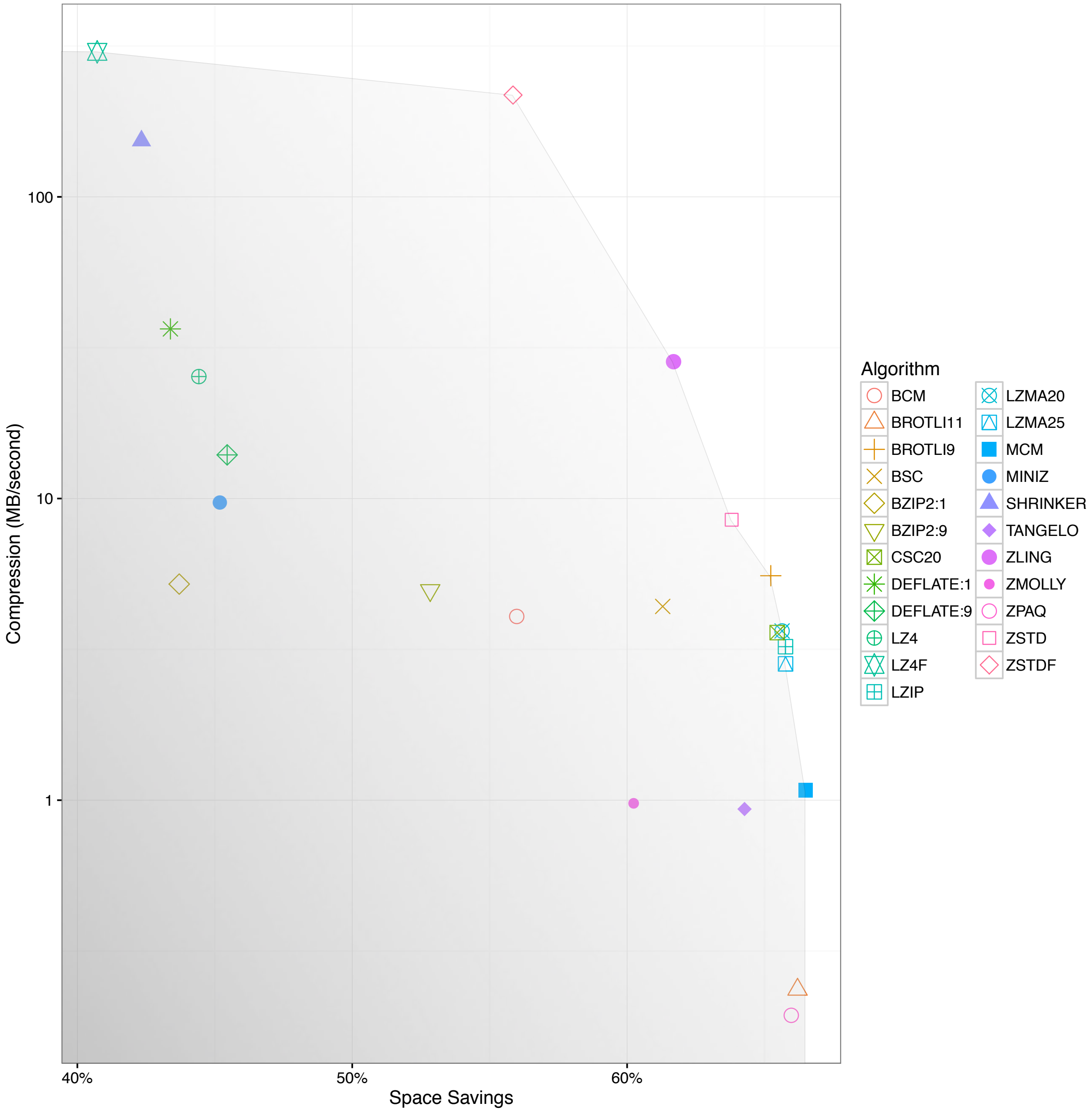


Figure 5.11: PCAP: Compression vs. Savings.

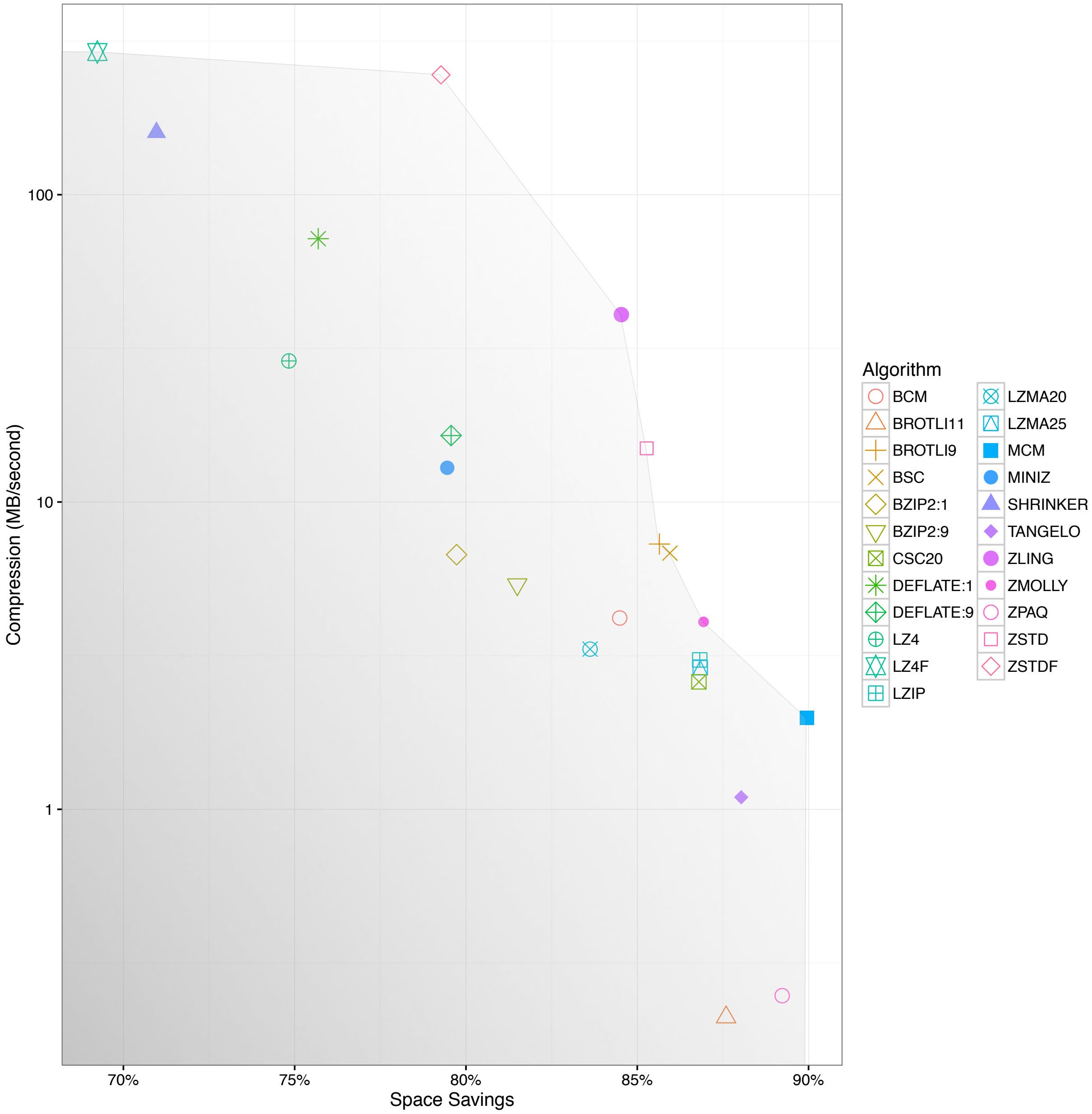


Figure 5.12: Bro: Compression vs. Savings.

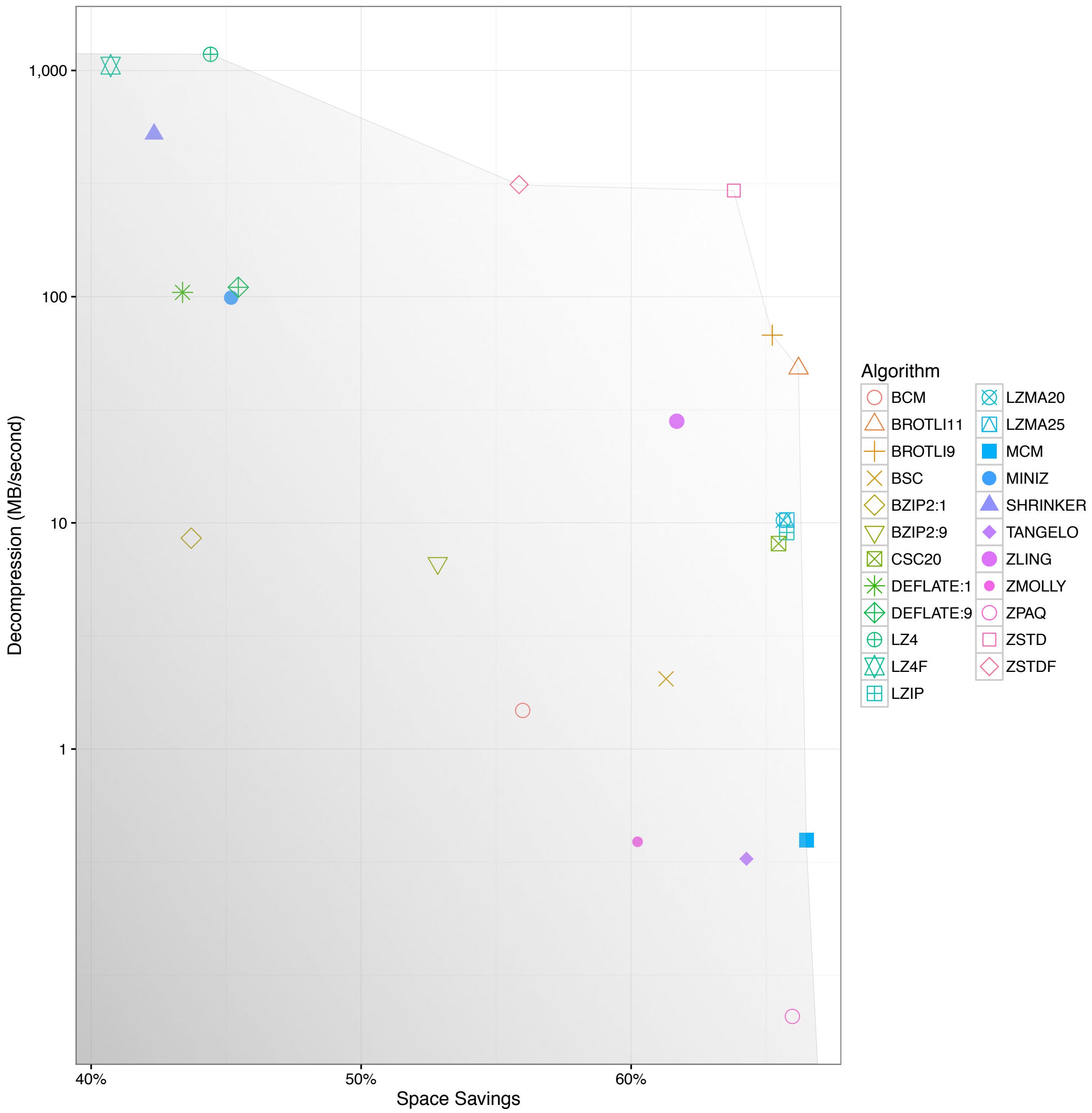


Figure 5.13: PCAP: Decompression vs. Savings.

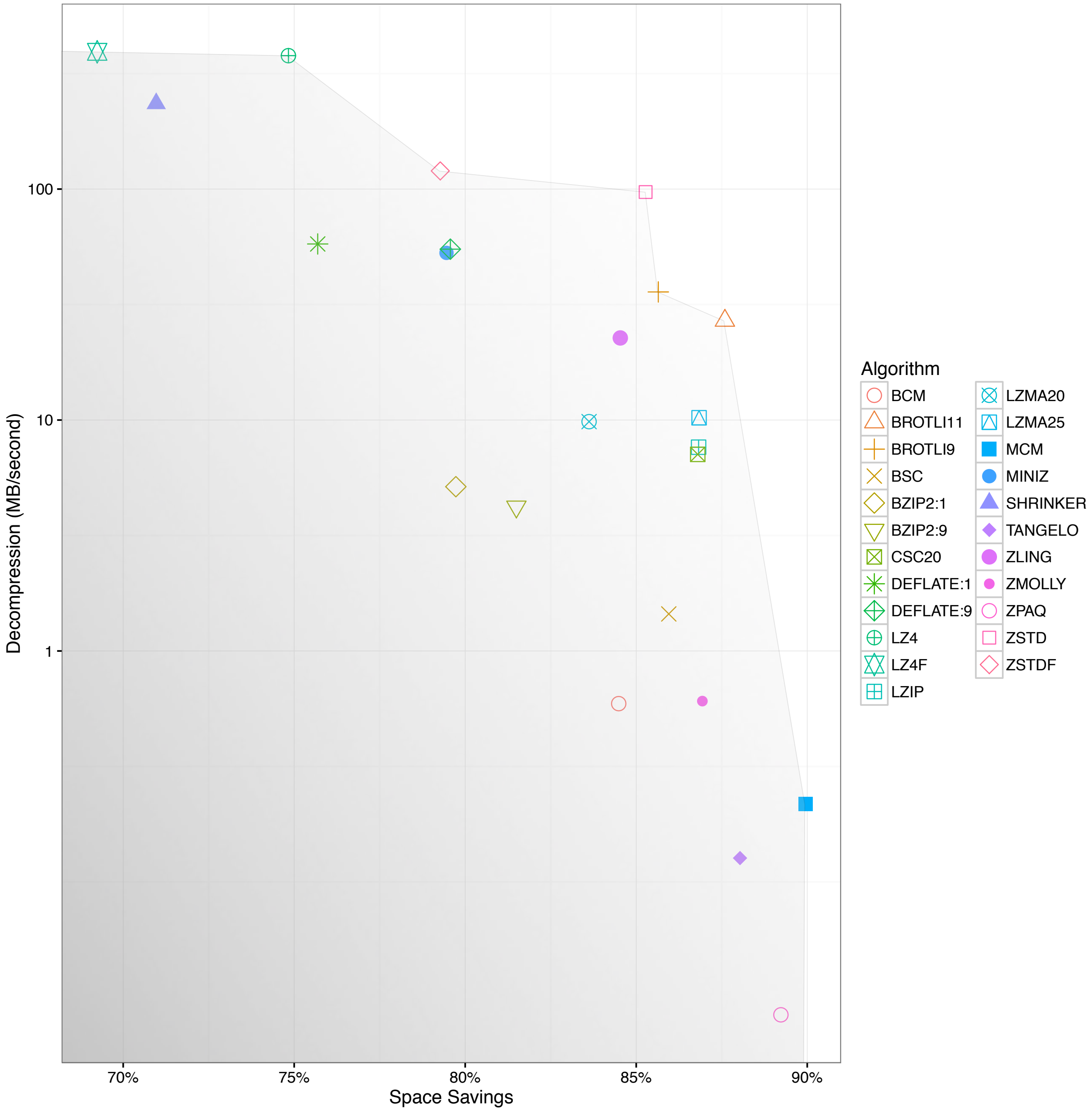


Figure 5.14: Bro: Decompression vs. Savings.

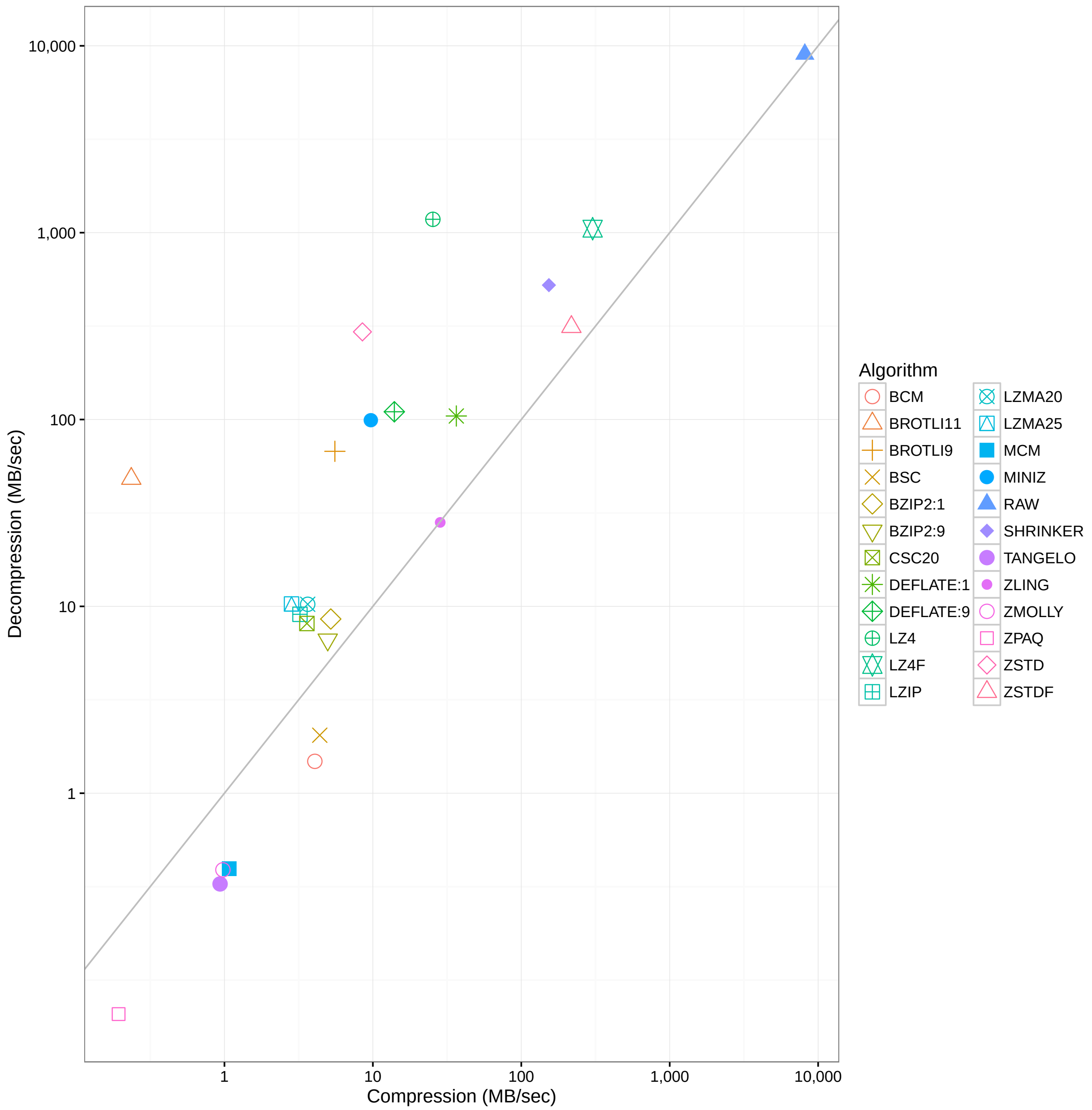


Figure 5.15: PCAP: Compression vs. Decompression.

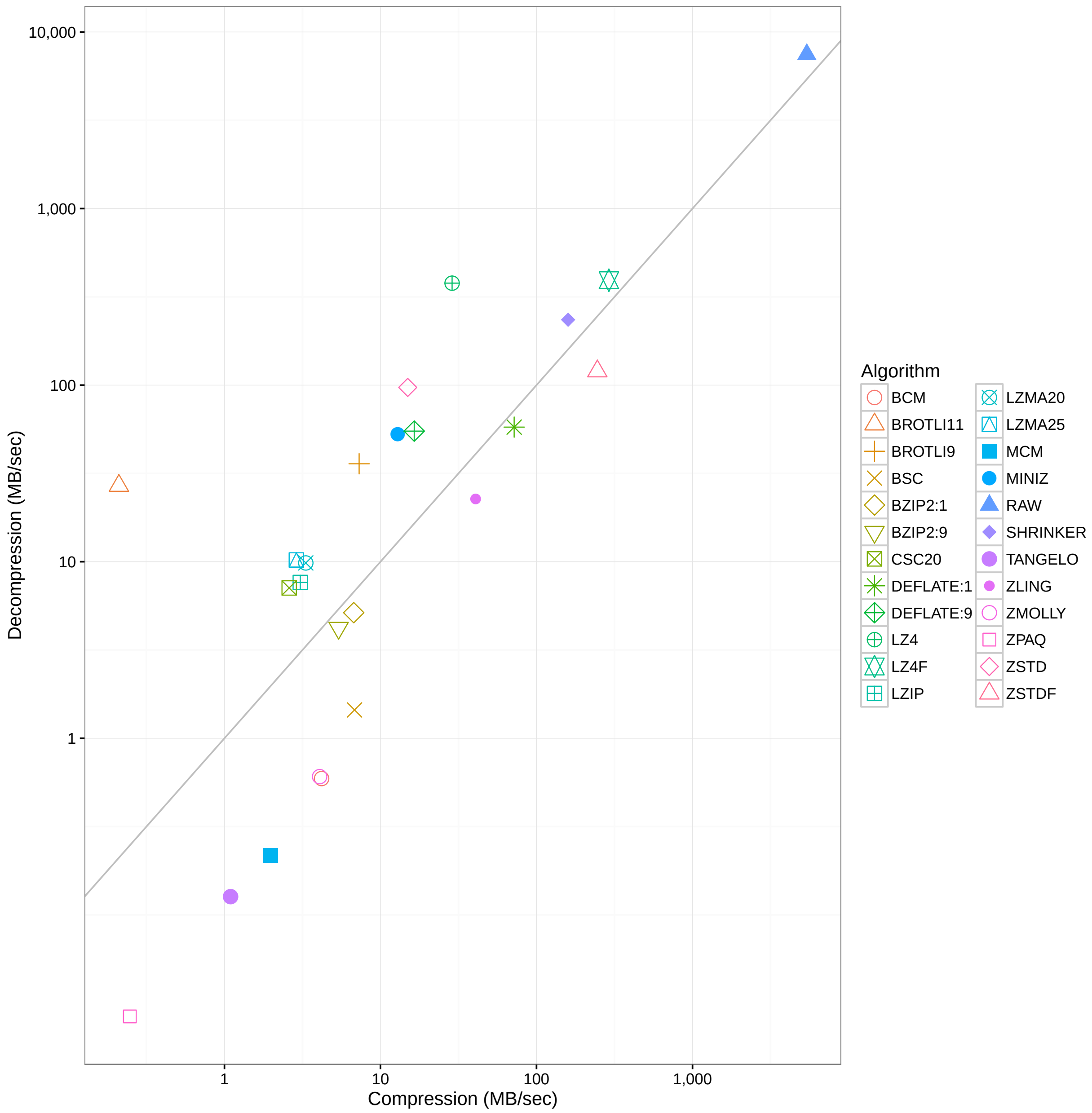


Figure 5.16: Bro: Compression vs. Decompression.

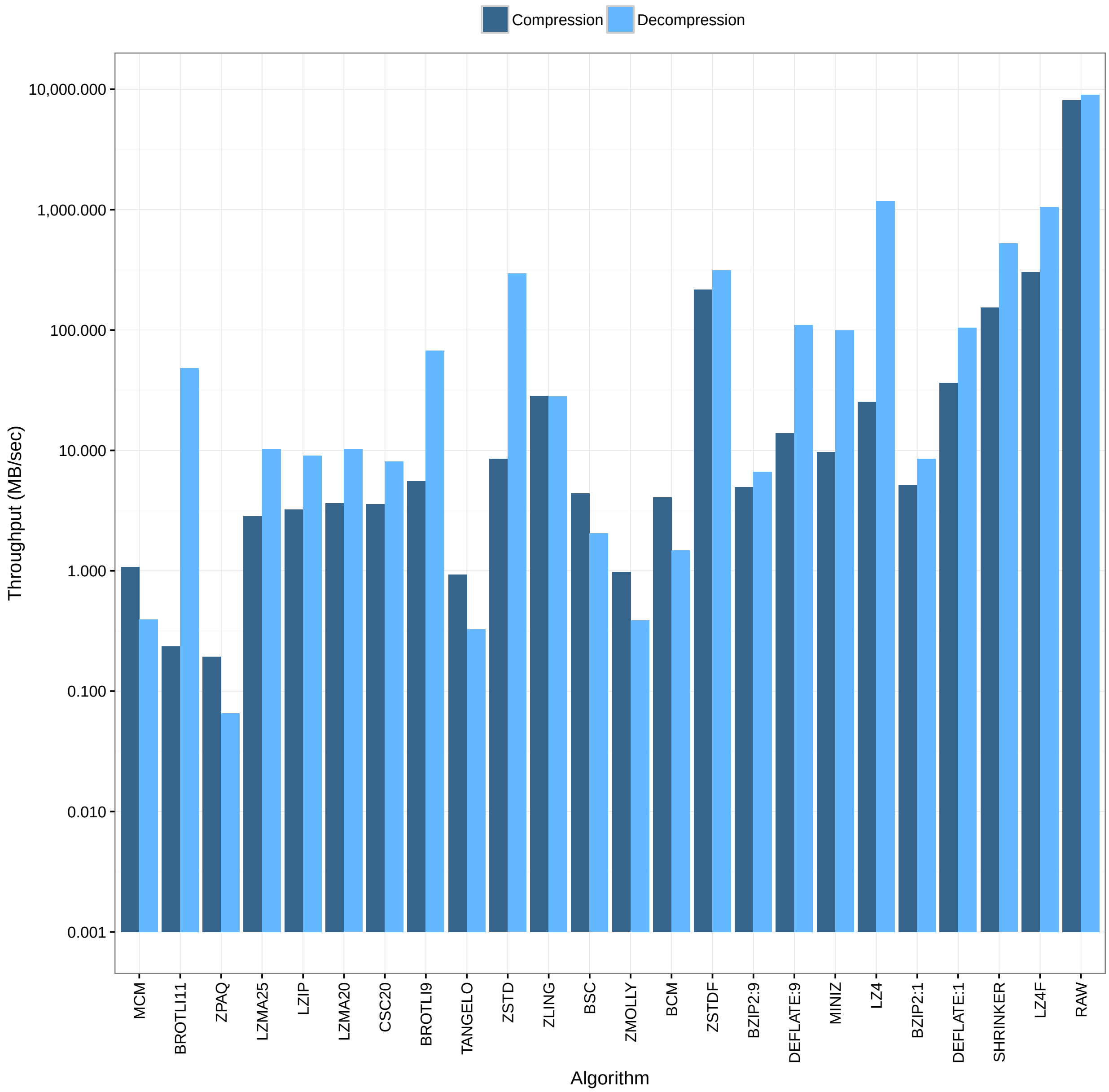


Figure 5.17: PCAP: Compression vs. Decompression. The algorithms on the x-axis are sorted with respect to Savings, with the most space-efficent algorithm on the left.

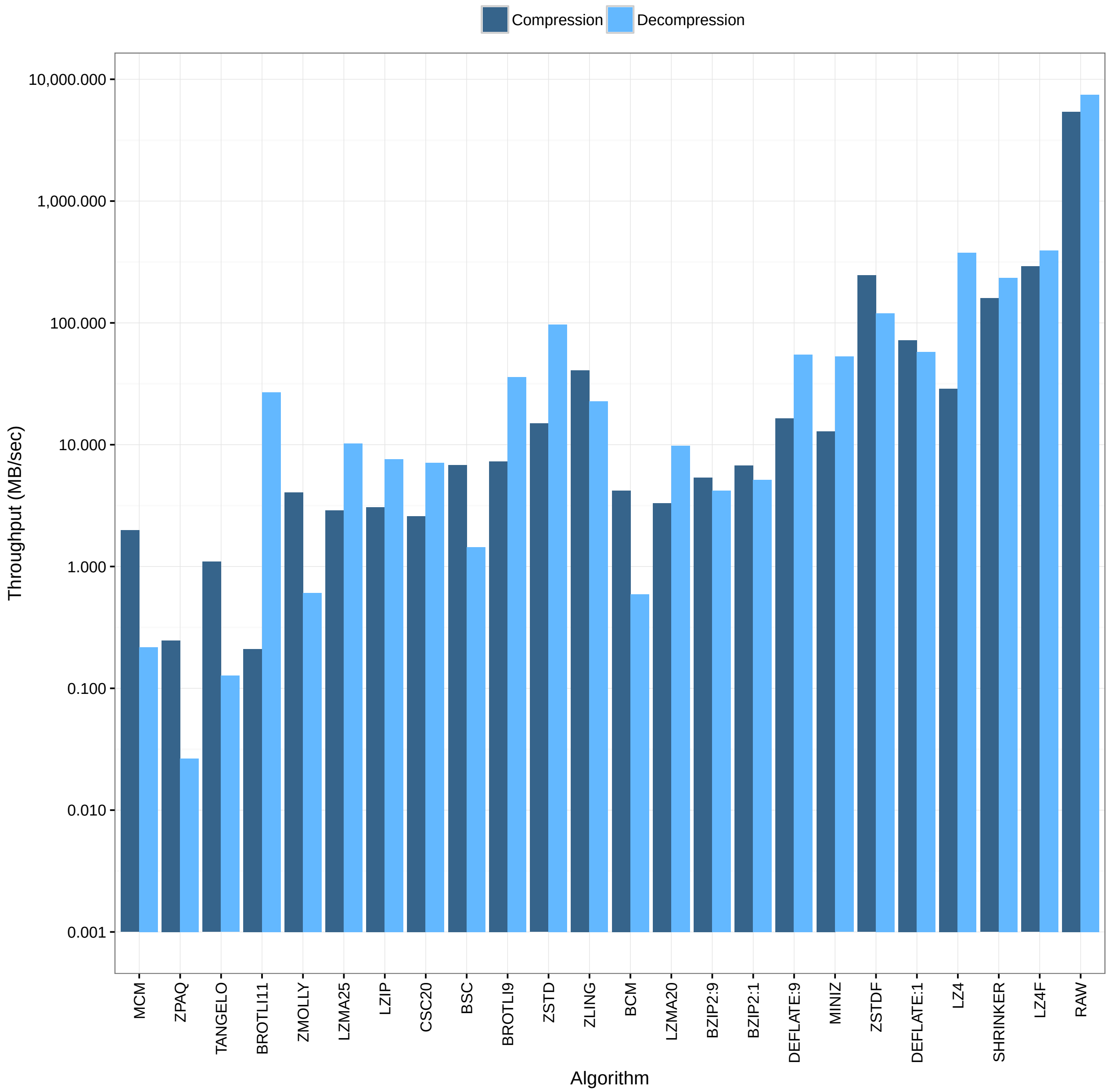


Figure 5.18: Bro: Compression vs. Decompression. The algorithms on the x-axis are sorted with respect to Savings, with the most space-efficent algorithm on the left.

Format	Archive	Index	Both
Bro	47%	90%	137%
PCAP	92%	4%	96%

Table 5.4: Storage overhead relative to the base data. The archive uses LZ4 [53] as compression algorithm.

5.6.2 Index Overhead

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%. Per §5.6.1, VAST also compresses the raw data using LZ4 [53] before storing it 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 5-tuple plus timestamps amounts to 4%. In total, VAST still occupies less space than the original data. We summarize the overhead in Table 5.4.

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. An extensive discussion about space consideration of bit vector encoding and compression schemes goes beyond the scope of this thesis, but holds promise for extensive tuning.

5.7 Summary

We evaluated VAST with respect to correctness (§5.2), throughput (§5.3), latency (§5.4), scaling (§5.5), and storage overhead (§5.6). To ensure correctness of operation, we embraced test-driven software engineering practices throughout development process. As of this writing, VAST ships with 6,700 lines of unit tests that check the system's basic building blocks (see §5.3).

Throughput represents a key performance metric when handling massive data volumes. Our measurements in §5.3 show that a single VAST instance can parse and index Bro logs at a rate of 100 K events/second, and PCAP packets at a rate of 260 K events/second. VAST's highly concurrent indexing architecture exhibits linear scaling gains, indicating that adding more CPU cores can achieve further improvements.

To support the highly interactive and iterative workflow of network forensics, a viable system must exhibit query latencies on the order of seconds. Our analysis in §5.4 shows that VAST can compute the full result set via index lookups in fractions of seconds such that analysts receive a taste of the results after about 1 second.

In §5.5, we examined VAST in a distributed setting and observed linear scaling with the number of nodes in the cluster. In terms of storage overhead, we analyzed the inherent space-time trade-off of various compression algorithms in §5.6.1. We chose LZ4 for compressing events at the archive because it proved the fastest algorithm, albeit at the cost of space savings. In §5.6.2, we established that storing PCAP packets requires, surprisingly, less space (about 96%) than the original input data, thanks to archive compression and small indexes. For Bro connection logs, we observed a storage overhead of 137% compared to the original data.

Overall, we found that our implementation meets our performance expectations. VAST can ingest hundreds of thousands of events per second and provides an interactive query experience with query latencies at the order of seconds over billions of events.

Chapter 6

Conclusion

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

Winston Churchill

This chapter concludes this thesis. We summarize key insights in §6.1 and sketch promising avenues for future work in §6.2.

6.1 Summary

Over the past twenty years, the process of criminal investigation has been revolutionized by modern forensic techniques, including DNA sequencing, mass spectrometry, and automated fingerprint identification. During the same period, unfortunately, the work of computing professionals who investigate cyber attacks has become increasingly frustrating and unwieldy. Available tools for real-time, root cause analysis of security incidents still remain crude, and fundamental breakthroughs in forensic techniques for cyber investigation are urgently needed (see §1).

This work centers around developing one such capability: a platform for forensic analysis that captures and retains a high-fidelity archive of cyber activity at the scale of an *entire network*, rather than a single host or network service. The system, in turn, supports interactive investigation of the activity—at a scale, speed, and degree of flexibility currently unavailable. Today, when security analysts attempt to reconstruct the sequence of events leading to a cyber compromise, they struggle to bring together enormous volumes of heterogeneous data, including alerts from intrusion detection systems, application logs from web or proxy servers, packet traces, DNS logs, and dozens of other diverse data types. Because these records describe events with widely differing semantics, analysts typically need to interact with multiple systems simultaneously, or else accept the loss of crucial semantic content when collectively aggregating all records inside a single Security Information and Event Management (SIEM) framework. These trade-offs lead to slow, ad-hoc, error-prone workflows—and, ultimately, to poor cybersecurity outcomes.

In this thesis we present the design and implementation of VAST (Visibility Across Space and Time), a platform that synthesizes powerful indexing technology with a distributed, entirely asynchronous system architecture (see §3). For indexing, VAST deploys a new form of high-level bitmap indexes (see §2) that enable fast data access, along with a set of corresponding algorithms that implement bitwise operations on compressed sequences in support of its type-rich data model. For scaling, VAST's architecture relies exclusively on the highly concurrent actor model, composing fine-granular tasks into parallel workflows that fully exploit the potential of modern many-core CPU platforms.

Our implementation follows the maxim of striving to achieve the highest degree of abstraction without sacrificing performance (see §4). Built on top of the C++ Actor Framework (CAF) [44], VAST constitutes a fully asynchronous message passing system that compiles down to execute native instructions, thereby avoiding performance penalties from intermediary abstract machines and/or stop-the-world garbage collection runtimes. Moreover, CAF hides network communication transparently, which allows for configuring flexible topologies at runtime. This pays off particularly when spreading computation over a cluster of commodity machines. As a result, VAST scales both inside a single machine and across clusters of physical systems.

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 (see §5). We release VAST as free open-source software under a permissive BSD license [190].

6.2 Outlook

This thesis laid the foundation for a host of fruitful future research. VAST holds promise for enabling exploration of new directions in improving large-scale forensic analyses at the level of entire networks.

6.2.1 Systems Challenges

From a system perspective, our prototype reveals numerous optimization considerations:

Novel encoding schemes. Software patents prevented us from exploring alternative bit vector encoding schemes at the inception of our project. In the meantime, several new encoding algorithms have emerged. A quantitative comparison with real-world data can give us more insight into which algorithm offers the best space-time trade-off. This has a direct impact on storage overhead and query latency.

Query optimization. VAST can further benefit from decades of research and experience in the database community concerning query optimization: for example, evaluating predicates according to the selectivity of the values in order to minimize the amount of operations required downstream in the execution. More broadly, VAST needs a cost-based optimizer to craft individual query plans which account for a variety of competing concerns.

Dynamic flow control. VAST consists of multiple components that communicate only via message passing. We implemented a form of flow control in the form of back pressure to prevent the system from "keeling over." Extending introspective monitoring to get a deep understanding of the current resource limits, and dynamically migrating bottleneck components to other machines poses an ambitious challenge.

6.2.2 Algorithmic Challenges

From a user perspective, detailed instrumentation would help to obtain traces of how analysts interact with the system. These traces can drive several research efforts:

Data structure tuning. Many VAST data structures have space-time trade-offs that depend on the nature of queries that analysts issue. Developing algorithms that dynamically adapt internal data structures (and migrate older instances) based on a given analyst's investigatory style hold promise for reducing query latency and improving storage overhead.

Prefetching. We envision developing algorithms to significantly enhance VAST's interactive performance by anticipating a given analyst's likely follow-on requests and prefetching meta-data (and possibly results) salient to those requests during the analyst's "think time". Doing so will enable VAST to instantly provide answers to the analyst if they indeed next make one of the anticipated requests.

Suggestion algorithms. By analyzing investigations conducted by a large number of analysts, we can identify *analysis patterns*: templates used by different analysts to drive their forensic exploration. Given such templates, we can then enhance VAST to provide *suggestions* to an analyst about possible next steps to augment their workflow.

Each of these efforts represents a new research area, unexplored to date in the literature, for enhancing the forensic process. The data provided from first experiences with VAST will play an instrumental role in empirically grounding this research in actual analyst behaviors; development of these algorithms holds promise for major improvement of the network forensics process at scale.

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, January 2005. ↑13

[2] ActorFoundry. `http://osl.cs.illinois.edu/software/actor-foundry`. Accessed December 27, 2015. ↑22

[3] Mark Adler. zlib. `http://www.zlib.net`. Accessed January 8, 2016. ↑105

[4] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2015. ↑16

[5] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986. ↑20

[6] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, June 1975. ↑76

[7] Akka. `http://akka.io`. Accessed December 3, 2015. ↑21, ↑22

[8] Mark Allman, Christian Kreibich, Vern Paxson, Robin Sommer, and Nicholas Weaver. Principles for Developing Comprehensive Network Visibility. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*, July 2008. ↑2, ↑11

[9] Witold Andrzejewski and Robert Wrembel. GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid. In *Proceedings of the Conference on Database and Expert Systems Applications (DEXA)*, 2010. ↑21

[10] G. Antoshenkov. Byte-aligned Bitmap Compression. In *Proceedings of the Conference on Data Compression (DCC)*, 1995. ↑34

[11] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2003. ↑13

[12] Joe Armstrong. Erlang——a Survey of the Language and its Industrial Applications. In *In Proceedings of the 9th Exhibitions and Symposium on Industrial Applications of Prolog*, INAP, 1996. ↑21

[13] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Department of Microelectronics and Information Technology, KTH, Sweden, 2003. ↑20, ↑22, ↑23, ↑59

[14] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004. ↑64

[15] Ricardo Baeza-Yates and Alejandro Salinger. Fast Intersection Algorithms for Sorted Sequences. In *Algorithms and Applications*, pages 45–61. Springer, 2010. ↑26

[16] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999. ↑76

[17] C. Bao, C. Huitema, M. Bagnulo, M. Boucadair, and X. Li. IPv6 Addressing of IPv4/IPv6 Translators. RFC 6052, Internet Engineering Task Force (IETF), October 2010. ↑77

[18] Jérémy Barbay, Alejandro López-Ortiz, and Tyler Lu. Faster Adaptive Set Intersections for Text Searching. In *Proceedings of the Conference on Experimental Algorithms (WEA)*, 2006. ↑26

[19] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indexes. In *Record of the ACM SIGFIDET Workshop on Data Description and Access*. ACM, November 1970. ↑25

[20] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. The Pyramid-technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998. ↑25

[21] Robin Berthier, Michel Cukier, Matti Hiltunen, Dave Kormann, Gregg Vesonder, and Dan Sheleheda. Nfsight: NetFlow-based Network Awareness Tool. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2010. ↑11

[22] Robert Beverly, Simson Garfinkel, and Greg Cardwell. Forensic Carving of Network Packets and Associated Data Structures. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 8:78–89, August 2011. ↑10

[23] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A Scripting Language for Large Scale Semistructured Data Analysis. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2011. ↑16

[24] Ledion Bitincka, Archana Ganapathi, and Steve Zhang. Experiences with Workload Management in Splunk. In *Proceedings of the Workshop on Management of Big Data Systems (MBDS)*, 2012. ↑13, ↑18

[25] Truls A. Bjørklund, Nils Grimsmo, Johannes Gehrke, and Øystein Torbjørnsen. Inverted Indexes vs. Bitmap Indexes in Decision Support Systems. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*, 2009. ↑33

[26] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970. ↑13, ↑17, ↑53

[27] Jasper Bongertz. The Needle in the Haystack. *Annual FIRST Conference on Computer Security Incident Handling*, June 2015. ↑11

[28] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762–772, October 1977. ↑76

[29] David A. Boyuka, II, Houjun Tang, Kushal Bansal, Xiaocheng Zou, Scott Klasky, and Nagiza F. Samatova. The Hyperdyadic Index and Generalized Indexing and Query with PIQUE. In *Proceedings of the Conference on Scientific and Statistical Database Management (SSDBM)*, 2015. ↑26

[30] The Bro Network Security Monitor. `http://www.bro.org`. Accessed January 13, 2016. ↑3, ↑17, ↑54

[31] Bundle, an embeddable compression library. `https://github.com/r-lyeh/bundle`. Accessed January 8, 2016. ↑105

[32] bzip2. `http://www.bzip.org`. Accessed January 8, 2016. ↑105

[33] C++ Actor Framework. `https://actor-framework.org`. Accessed May 3, 2016. ↑84

[34] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011. ↑60

[35] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams: A New Class of Data Management Applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002. ↑13

[36] Rick Cattell. Scalable SQL and NoSQL Data Stores. *SIGMOD Record*, 39(4):13, 2010. ↑14

[37] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A Structured English Query Language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. ACM, 1974. ↑12

[38] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better Bitmap Performance with Roaring Bitmaps. *CoRR*, abs/1402.6407, 2014. ↑27, ↑33

[39] Chee-Yong Chan and Yannis E. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1998. ↑31, ↑32, ↑33, ↑34, ↑36

[40] Chee-Yong Chan and Yannis E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999. ↑29, ↑31, ↑33, ↑36, ↑38

[41] Chee-Yong Chan and Yannis E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. *SIGMOD Record*, 28(2):215–226, June 1999. ↑32

[42] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008. ↑15

[43] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. Revisiting Actor Programming in C++. *Computer Languages, Systems & Structures*, January 2016. (in press). ↑24

[44] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proceedings of the ACM Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*, October 2013. ↑22, ↑23, ↑118

[45] Mathieu Chartier. mcm compressor: context mixing + lzp. `https://github.com/mathieuchartier/mcm`. Accessed January 8, 2016. ↑105

[46] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, March 1997. ↑12, ↑14

[47] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the ACM Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE!)*, 2015. ↑22

[48] CMake. `https://cmake.org`. Accessed May 3, 2016. ↑84

[49] Ira Cohen, Steve Zhang, Moises Goldszmidt, Julie Symons, Terence Kelly, and Armando Fox. Capturing, Indexing, Clustering, and Retrieving System History. *SIGOPS Operating Systems Review*, 39:105–118, October 2005. ↑11

[50] Jeffrey Cohen, John Eshleman, Brian Hagenbuch, Joy Kent, Christopher Pedrotti, Gavin Sherry, and Florian Waas. Online Expansion of Large-scale Data Warehouses. *Proceedings of the VLDB Endowment*, 4(12), 2011. ↑13, ↑14

[51] M. I. Cohen, D. Bilby, and G. Caronni. Distributed Forensics and Incident Response in the Enterprise. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 8:101–110, August 2011. ↑10

[52] Alessandro Colantonio and Roberto Di Pietro. CONCISE: Compressed 'n' Composable Integer Set. *Information Processing Letters*, 110(16):644–650, July 2010. ↑34

[53] Yann Collet. LZ4: Extremely Fast Compression algorithm. `http://www.lz4.org`. Accessed January 8, 2016. ↑105, ↑115

[54] Yann Collet. Zstandard. `http://www.zstd.net`. Accessed January 8, 2016. ↑105

[55] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. MapReduce Online. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010. ↑16

[56] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005. ↑13

[57] Fabian Corrales, David Chiu, and Jason Sawin. Variable Length Compression for Bitmap Indices. In *Proceedings of the Conference on Database and Expert Systems Applications (DEXA)*, 2011. ↑34

[58] data-shrinker: a very light and fast compression program with acceptable ratio. `https://code.google.com/p/data-shrinker/`, shrinker. Accessed January 2, 2016. ↑105

[59] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, 2004. ↑15, ↑16

[60] François Deliège and Torben Bach Pedersen. Position List Word Aligned Hybrid: Optimizing Space and Performance for Compressed Bitmaps. In *Proceedings of the Conference on Extending Database Technology (EDBT)*, 2010. ↑33, ↑34

[61] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive Set Intersections, Unions, and Differences. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2000. ↑26

[62] Travis Desell and Carlos A. Varela. SALSA Lite: A Hash-Based Actor Runtime for Efficient Local Concurrency. In *Concurrent Objects and Beyond*, volume 8665 of *Lecture Notes in Computer Science*, pages 144–166. Springer, 2014. ↑22

[63] Antonio Diaz Diaz. Lzip. `http://www.nongnu.org/lzip/lzip.html`. Accessed January 8, 2016. ↑105

[64] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, September 2010. ↑60

[65] Fredton Doan, David Chiu, Brasil Perez Lukes, Jason Sawin, Gheorghi Guzun, and Guadalupe Canahuate. Dynamic Bitmap Index Recompression Through Workload-based Optimizations. In *Proceedings of the ACM International Database Engineering and Applications Symposium (IDEAS)*, 2013. ↑34

[66] Walter J. Doherty and Ahrvind J. Thadani. The Economic Value of Rapid Response Time. *IBM*, (GE20-0752), November 1982. ↑5

[67] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. Operational Experiences with High-volume Network Intrusion Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004. ↑64

[68] Josiah Dykstra and Alan T. Sherman. Acquiring Forensic Evidence from Infrastructure-as-a-Service Cloud Computing: Exploring and Evaluating Tools, Trust, and Techniques. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 9:90–98, 2012. ↑10

[69] ElasticSearch. `https://www.elastic.co/products/elasticsearch`. Accessed December 6, 2015. ↑15, ↑60, ↑76

[70] Elixir. `http://elixir-lang.org`. Accessed December 27, 2015. ↑22

[71] etcd. `https://github.com/coreos/etcd`. Accessed December 31, 2015. ↑43

[72] W. Fernandez de la Vega, A. M. Frieze, and M. Santha. Average-Case Analysis of the Merging Algorithm of Hwang and Lin. *Algorithmica*, 22(4):483–489, 1998. ↑26

[73] Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, May 2002. ↑15

[74] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. HyperLogLog: the Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Proceedings of the International Conference of Analysis of Algorithms (AOFA)*, pages 127–146, 2007. ↑13

[75] Francesco Fusco, Xenofontas Dimitropoulos, Michail Vlachos, and Luca Deri. pcapindex: An index for network packet traces with legacy compatibility. *SIGCOMM Computer Communications Review*, 42(1):47–53, January 2012. ↑17

[76] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. NET-FLi: On-the-fly Compression, Archiving and Indexing of Streaming Network Traffic. *Proceedings of the VLDB Endowment*, 3(1-2):1382–1393, September 2010. ↑18, ↑33, ↑34

[77] Francesco Fusco, Michael Vlachos, Xenofontas Dimitropoulos, and Luca Deri. Indexing Million of Packets per Second using GPUs. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2013. ↑18, ↑21

[78] Francesco Fusco, Michail Vlachos, and Xenofontas Dimitropoulos. RasterZip: Compressing Network Monitoring Data with Support for Partial Decompression. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2012. ↑48, ↑99, ↑102

[79] Roland Färber. *Römische Gerichtsorte: Räumliche Dynamiken von Jurisdiktion im Imperium Romanum*, volume 68 of *Vestigia*. C.H. Beck, November 2014. ↑3

[80] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*. Springer, 2004. ↑23, ↑24

[81] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, June 1998. ↑25

[82] Simson Garfinkel. Digital Forensics Research: The Next 10 Years. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 7:64–73, 2010. ↑11

[83] Simson Garfinkel. Lessons learned writing digital forensics tools and managing a 30TB digital evidence corpus. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 9:80–89, 2012. ↑10

[84] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, 9(11):40:40–40:54, November 2011. ↑44, ↑63

[85] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. *ACM SIGOPS Operating Systems Review*, 37(5):29–43, 2003. ↑15, ↑16, ↑60

[86] Paul Giura and Nasir Memon. NetStore: An Efficient Storage Infrastructure for Network Forensics and Monitoring. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 277–296, 2010. ↑18

[87] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003. ↑13

[88] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994. ↑29

[89] Ilya Grebnov. libbsc: High performance block-sorting data compression library. `http://libbsc.com`. Accessed January 8, 2016. ↑105

[90] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2015. ↑63

[91] Ashish Gupta, Fan Yang, Jason Govig, Adam Kirsch, Kelvin Chan, Kevin Lai, Shuo Wu, Sandeep Govind Dhoot, Abhilash Rajesh Kumar, Ankur Agiwal, Sanjay Bhansali, Mingsheng Hong, Jamie Cameron, Masood Siddiqi, David Jones, Jeff Shute, Andrey Gubarev, Shivakumar Venkataraman, and Divyakant Agrawal. Mesa: Geo-replicated, Near Real-time, Scalable Data Warehousing. *Proceeding of the VLDB Endowment*, 7(12):1259–1270, August 2014. ↑15

[92] G. Guzun, G. Canahuate, D. Chiu, and J. Sawin. A Tunable Compression Framework for Bitmap Indices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, March 2014. ↑34

[93] Gheorghi Guzun and Guadalupe Canahuate. Performance Evaluation of Word-Aligned Compression Methods for Bitmap Indices. *Knowledge and Information Systems*, pages 1–28, 2015. ↑33, ↑34

[94] Hadoop. `http://hadoop.apache.org`. Accessed December 6, 2015. ↑13

[95] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. Processing a Trillion Cells Per Mouse Click. *Proceedings of the VLDB Endowment*, 5(11):1436–1446, July 2012. ↑14

[96] Stefan Heule, Marc Nunkesser, and Alexander Hall. HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm. In *Proceedings of the Conference on Extending Database Technology (EDBT)*, pages 683–692. ACM, 2013. ↑13

[97] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the IJCAI*, 1973. ↑20, ↑63

[98] Erik Hjelmvik. Hands-on Network Forensics. *Annual FIRST Conference on Computer Security Incident Handling*, June 2015. ↑11

[99] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978. ↑20, ↑63

[100] A.S. Hornby. *Oxford Advanced Learner's Dictionary of Current English.* Oxford University Press, 6 edition, 2000. ↑2

[101] Bill Horne. Collecting, Analyzing and Responding to Enterprise Scale DNS Events. *Annual FIRST Conference on Computer Security Incident Handling*, June 2015. ↑11

[102] Hughes, Evan and Somayaji, Anil. Towards Network Awareness. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2005. ↑11

[103] F. K. Hwang and S. Lin. Optimal Merging of 2 Elements with N Elements. *Acta Informatica*, 1(2):145–158, June 1971. ↑26

[104] Franz Faerber Ingo Müller, Cornelius Ratsch. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *Proceedings of the Conference on Extending Database Technology (EDBT)*, pages 283–294, March 2014. ↑73

[105] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2007. ↑16

[106] ISO/IEC. Information technology – Database languages – SQL. Standard 9075:2011, 2011. ↑12

[107] ISO/IEC. Information technology – Microprocessor Systems – Floating-Point arithmetic. Standard 60559:2011, 2011. ↑70

[108] Apache Kafka. `http://kafka.apache.org`. Accessed January 13, 2016. ↑54, ↑60, ↑65

[109] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In *Parallel Programming using C++*, pages 175–213. MIT Press, 1996. ↑22

[110] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. BlowFish: Dynamic Storage-Performance Tradeoff in Data Stores. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2016. ↑60

[111] Ryan King. Announcing Snowflake. `https://blog.twitter.com/2010/announcing-snowflake`, 2010. Accessed January 2, 2016. ↑44

[112] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Pearson Education, 1998. ↑25, ↑26

[113] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *Journal of the Society for Industrial and Applied Mathematics*, 6(2):323–350, 1977. ↑76

[114] Christian Kreibich, Nicholas Weaver, Boris Nechaev, and Vern Paxson. Netalyzr: Illuminating the Edge Network. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2010. ↑44, ↑63

[115] Paul Krizak. Log Analysis and Event Correlation Using Variable Temporal Event Correlator (VTEC). In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2010. ↑11

[116] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The Vertica Analytic Database: C-store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, August 2012. ↑15

[117] Paul J. Leach, Michael Mealling, and Rich Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122, Internet Engineering Task Force (IETF), July 2005. ↑45

[118] George Lee, Jimmy Lin, Chuang Liu, Andrew Lorek, and Dmitriy Ryaboy. The Unified Logging Infrastructure for Data Analytics at Twitter. *Proceedings of the VLDB Endowment*, 5(12):1771–1780, 2012. ↑14

[119] Jihyung Lee, Sungryoul Lee, Junghee Lee, Yung Yi, and KyoungSoo Park. FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015. ↑11, ↑17

[120] Daniel Lemire and Leonid Boytsov. Decoding Billions of Integers per Second Through Vectorization. *CoRR*, abs/1209.2137, 2012. ↑33

[121] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting Improves Word-aligned Bitmap Indexes. *Data & Knowledge Engineering*, 69(1):3–28, January 2010. ↑26, ↑34

[122] Zhang Li. libzling: fast and niubility compression library. `https://github.com/richox/libzling`. Accessed January 8, 2016. ↑105

[123] Zhang Li. zmolly: PPM compressor with high compression ratio. `https://github.com/richox/zmolly`. Accessed January 8, 2016. ↑105

[124] libprocess. `https://github.com/3rdparty/libprocess`. Accessed December 26, 2015. ↑22

[125] Julie Beth Lovins. Development of a Stemming Algorithm. *Mechanical Translation and Computational Linguistics*, 11:22—31, 1968. ↑76

[126] Lucene. `https://lucene.apache.org`. Accessed December 4, 2015. ↑15, ↑76, ↑77

[127] Lempel–Ziv–Markov chain algorithm. https://en.wikipedia.org/wiki/Lempel-Ziv-Markov_chain_algorithm. Accessed January 8, 2016. ↑105

[128] M57-Patents Scenario PCAP Trace. http://digitalcorpora.org/corpora/scenarios/m57-patents-scenario. Accessed May 3, 2016. ↑89

[129] Matt Mahoney. ZPAQ: Incremental Journaling Backup Utility and Archiver. http://mattmahoney.net/dc/zpaq.html. Accessed January 8, 2016. ↑105

[130] Gregor Maier, Robin Sommer, Holger Dreger, Anja Feldmann, Vern Paxson, and Fabian Schneider. Enriching Network Security Analysis with Time Travel. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols For Computer Communications (SIGCOMM)*, August 2008. ↑17, ↑89

[131] Miguel A. Martínez-Prieto, Nieves Brisaboa, Rodrigo Cánovas, Francisco Claude, and Gonzalo Navarro. Practical Compressed String Dictionaries. *Information Systems*, 56:73–108, March 2016. ↑73

[132] Manish Mehta and David J. DeWitt. Data Placement in Shared-Nothing Parallel Database Systems. *The VLDB Journal*, 6(1):53–72, 1997. ↑43

[133] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, September 2010. ↑15, ↑48

[134] miniz: Single C source file Deflate/Inflate compression library with zlib-compatible API, ZIP archive reading/writing, PNG writing. https://code.google.com/p/miniz/. Accessed January 8, 2016. ↑105

[135] Scott Moeller, Avinash Sridharan, Bhaskar Krishnamachari, and Omprakash Gnawali. Routing Without Routes: The Backpressure Collection Protocol. In *Proceedings of International Conference on Information Processing in Sensor Networks (IPSN)*, 2010. ↑64

[136] Curt Monash. Splunk and inverted-list indexing. http://www.dbms2.com/2014/03/06/splunk-and-inverted-list-indexing/. Accessed November 15, 2015. ↑18

[137] mongoDB. http://www.mongodb.org. Accessed December 4, 2015. ↑13

[138] MySQL. http://www.mysql.com. Accessed December 4, 2015. ↑13

[139] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig Latin: A Not-So-Foreign Language for Data Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008. ↑16

[140] Patrick O'Neil and Dallan Quass. Improved Query Performance With Variant Indexes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1997.

[141] Patrick E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the International Workshop on High Performance Transaction Systems*, 1987. ↑26

[142] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2014. ↑43

[143] OpenCL: The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl`. Accessed December 4, 2015. ↑21

[144] ORACLE. `http://www.oracle.com`. Accessed December 4, 2015. ↑13

[145] Giuseppe Ottaviano and Rossano Venturini. Partitioned Elias-Fano Indexes. In *Proceedings of ACM Conference on Research & Development in Information Retrieval (SIGIR)*, 2014. ↑33

[146] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast File System. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, August 2013. ↑16

[147] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999. ↑3, ↑17, ↑105

[148] Robert Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277–298, 2005. ↑16

[149] Martin F. Porter. Readings in information retrieval. chapter An Algorithm for Suffix Stripping, pages 313–316. Morgan Kaufmann Publishers Inc., 1997. ↑76

[150] PostgreSQL. `http://www.postgresql.org`. Accessed December 4, 2015. ↑13

[151] Pulsar. `http://docs.paralleluniverse.co/pulsar/`. Accessed December 27, 2015. ↑22

[152] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2013. ↑13

[153] Ariel Rabkin and Randy Katz. Chukwa: A system for reliable large-scale log collection. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2010. ↑11

[154] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *USENIX Conference on File and Storage Technologies (FAST)*, February 2015. ↑60

[155] Reactive Streams. `http://www.reactive-streams.org`. Accessed April 11, 2016. ↑64

[156] Redis. `http://redis.io`. Accessed December 4, 2015. ↑13

[157] Irving S Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960. ↑60

[158] Frederick Reiss, Kurt Stockinger, Kesheng Wu, Arie Shoshani, and Joseph M. Hellerstein. Enabling Real-Time Querying of Live and Historical Stream Data. In *Proceedings of the Conference on Scientific and Statistical Database Management (SSDBM)*, 2007. ↑14

[159] Mark Reith, Clint Carr, and Gregg Gunsch. An Examination of Digital Forensic Models. *International Journal of Digital Evidence*, 1(3):1–12, 2002. ↑3

[160] Riak. `http://www.basho.com`. Accessed December 4, 2015. ↑13

[161] Rodrigo Rodrigues and Barbara Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *Proceedings of the International Conference on Peer-to-Peer Systems (IPTPS)*, 2005. ↑60

[162] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Optimizing Candidate Check Costs for Bitmap Indices. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*, 2005. ↑29, ↑102

[163] SAP HANA. `http://hana.sap.com`. Accessed December 1, 2015. ↑14

[164] Andreas Schmidt and Mirko Beine. A Concept for a Compression Scheme of Medium-Sparse Bitmaps. In *Procceedings of the The Third International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, pages 192–195, 2011. ↑34

[165] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of Inverted Indexes For Fast Query Evaluation. In *Proceedings of ACM Conference on Research & Development in Information Retrieval (SIGIR)*, 2002. ↑33

[166] Malte Schwarzkopf. *Operating System Support for Warehouse-Scale Computing*. PhD thesis, University of Cambridge Computer Laboratory, 2015. ↑14

[167] Frank Sear. *Dictionary of Art*, volume 11. Grove, 1996. ↑3

[168] Matt Selsky and Daniel Medina. GULP: A Unified Logging Architecture for Authentication Data. In *Proceedings of the USENIX Large Installation System Administration Conference (LISA)*, 2005. ↑11

[169] Bilal Shebaro and Jedidiah R. Crandall. Privacy-preserving Network Flow Recording. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 8:90–100, August 2011. ↑10

[170] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010. ↑16, ↑60

[171] Abe Singer. Building a Logging Infrastructure. *FIRST Technical Colloquium*, October 2005. ↑11

[172] Rishi Rakesh Sinha and Marianne Winslett. Multi-resolution Bitmap Indexes for Scientific Data. *ACM Transactions on Database Systems (TODS)*, 32(3), August 2007. ↑18, ↑30, ↑54

[173] Apache Solr. `http://lucene.apache.org/solr`. Accessed February 19, 2016. ↑15, ↑60

[174] Kurt Stockinger, John Cieslewicz, Kesheng Wu, Doron Rotem, and Arie Shoshani. Using Bitmap Index for Joint Queries on Structured and Text Data. In *New Trends in Data Warehousing and Data Analysis*, volume 3. Springer, 2009. ↑73

[175] Kurt Stockinger, Kesheng Wu, and Arie Shoshani. Evaluation Strategies for Bitmap Indices with Binning. In *Proceedings of the Conference on Database and Expert Systems Applications (DEXA)*, volume 3180, 2004. ↑26

[176] M. Stonebraker, D.J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, et al. C-Store: A Column-oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2005. ↑15, ↑48

[177] Michael Stonebraker. SQL Databases V. NoSQL Databases. *Communications of the ACM*, 53(4):10–11, April 2010. ↑14

[178] Zoltan Szabadka. Introducing Brotli: a New Compression Algorithm for the Internet. `http://google-opensource.blogspot.com/2015/09/introducing-brotli-new-compression.html`. Accessed January 8, 2016. ↑105

[179] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *Communications Surveys & Tutorials*, 14(1):131–155, 2012. ↑13

[180] Leandros Tassiulas and Anthony Ephremides. Stability Properties of Constrained Queueing Systems and Scheduling Policies for Maximum Throughput in Multihop Radio Networks. *IEEE Transactions on Automatic Control*, 37(12):1936–1948, 1992. ↑64

[181] Teryl Taylor, Scott E. Coull, Fabian Monrose, and John McHugh. Toward Efficient Querying of Compressed Network Payloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012. ↑11, ↑18, ↑48, ↑73

[182] Theron. `http://www.theron-library.com`. Accessed December 27, 2015. ↑22

[183] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive – A Warehousing Solution Over a Map-Reduce Framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009. ↑13, ↑14, ↑16

[184] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data Warehousing and Analytics Infrastructure at Facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2010. ↑14

[185] John R. Vacca. *Computer Forensics: Computer Crime Scene Investigation*. Charles River Media, Inc., 2005. ↑3

[186] Matthias Vallentin. Benchmark and Visualization of Various Compression Algorithms. `https://github.com/mavam/compbench`. Accessed January 6, 2016. ↑105

[187] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The NIDS Cluster: Scalably Stateful Network Intrusion Detection on Commodity Hardware. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2007. ↑17

[188] Sebastiaan J. van Schaik and Oege de Moor. A Memory Efficient Reachability Data Structure Through Bit Vector Compression. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 913–924, 2011. ↑34

[189] Panos Vassiliadis. A Survey of Extract-Transform-Load Technology. *International Journal of Data Warehousing & Mining*, 5(3):1–27, July 2009. ↑12

[190] VAST: Visibility Across Space and Time. `http://vast.io`. Accessed March 30, 2016. ↑118

[191] Arun Viswanathan, Alefiya Hussain, Jelena Mirkovic, Stephen Schwab, and John Wroclawski. A Semantic Framework for Data Analysis in Networked Systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011. ↑11

[192] F. Tobagi W. Noureddine. Selective Back-pressure in Switched Ethernet LANs. In *Proceedings of the Global Telecommunications Conference (Globecom)*, 1999. ↑64

[193] Robert J. Walls, Brian N. Levine, Marc Liberatore, and Clay Shields. Effective Digital Forensics Research is Investigator-Centric. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*, 2011. ↑11

[194] Daniel Walnyckya, Ibrahim Baggilia, Andrew Marringtonb, Jason Moorea, and Frank Breitingera. Network and device forensic analysis of Android social-messaging applications. *Digital Investigation: The International Journal of Digital Forensics & Incident Response*, 14:77–84, 2015. ↑11

[195] Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, and Ulf Leser. State-of-the-art in String Similarity Search and Join. *SIGMOD Record*, 43(1):64–76, May 2014. ↑77

[196] Wei Wang and Thomas E. Daniels. Building Evidence Graphs for Network Forensics Analysis. In *Proceedings of the Computer Security Applications Conference (ACSAC)*, 2005. ↑11

[197] Hakim Weatherspoon and John Kubiatowicz. Erasure Coding vs. Replication: a Quantitative Comparison. In *Proceedings of the International Conference on Peer-to-Peer Systems (IPTPS)*, 2002. ↑60

[198] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012. ↑16

[199] Harry K. T. Wong, Hsiu-Fen Liu, Frank Olken, Doron Rotem, and Linda Wong. Bit Transposed Files. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1985. ↑26, ↑34, ↑35, ↑86

[200] Kesheng Wu. FastBit: an efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 16:556–560, 2005. ↑27

[201] Kesheng Wu, Ekow Otoo, and Arie Shoshani. On the Performance of Bitmap Indices for High Cardinality Attributes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2004. ↑26, ↑34

[202] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. A Performance Comparison of Bitmap Indexes. In *Proceedings of the Conference on Information and Knowledge Management (CIKM)*, 2001. ↑33, ↑34

[203] Lili Wu, Roshan Sumbaly, Chris Riccomini, Gordon Koo, Hyung Jin Kim, Jay Kreps, and Sam Shah. Avatara: OLAP for Web-Scale Analytics Products. *Proceedings of the VLDB Endowment*, 5(12):1874–1877, 2012. ↑13, ↑14

[204] Ming-Chuan Wu. Query Optimization for Selections Using Bitmaps. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1999.

[205] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A Real-time Analytical Data Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2014. ↑15

[206] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. Beehive: Large-scale Log Analysis for Detecting Suspicious Activity in Enterprise Networks. In *Proceedings of the Computer Security Applications Conference (ACSAC)*, 2013. ↑11

[207] Lei Ying, Sanjay Shakkottai, Aneesh Reddy, and Shihuan Liu. On Combining Shortest-path and Back-pressure Routing over Multihop Wireless Networks. *IEEE/ACM Transactions on Networking*, 19(3), June 2011. ↑64

[208] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language. In *Proceedings of the USENIX Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, 2008. ↑13

[209] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 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. ↑13, ↑16

[210] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013. ↑16

Appendix A

Multi-Component Range Queries

In §2.4.4 we mention the algorithm RangeEval-Opt [39], which performs a range query over a multi-component index. It operates more efficiently than its predecessor RangeEval [140], hence the Opt suffix. There exist further improvements to RangeEval-Opt which prune subtrees when an identifier set represents the full domain or the empty set [204]. The algorithm specifically works with range-encoded identifier sets. To get an intuitive understanding of how it works, we show in Figure A.1 an operator graph which reflects the evaluations for the specific value $x = 1337$ in uniform base 10. The algorithm performs a series of unions and intersections to arrive at the final result. Algorithm A.1 displays the complete algorithmic operation, adapted to fit in our notation which unifies inverted and bitmap indexes.

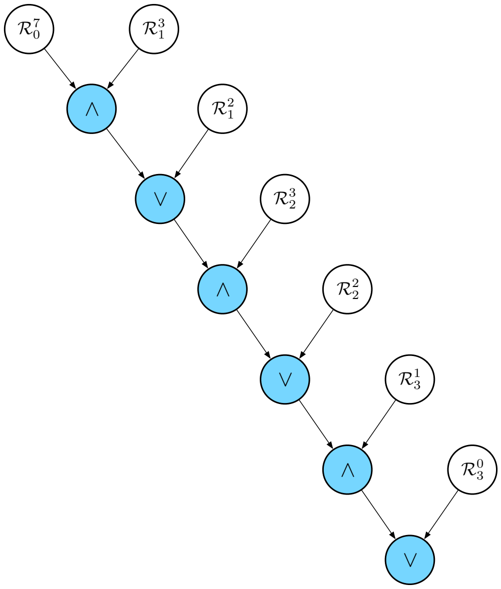


Figure A.1: Evaluation tree for the algorithm RangeEval-Opt (see Algorithm A.1) for value $x = 1337$ in uniform base 10.

Precondition:

I: an index with k components.
x: a value from the domain of I.
β: a well-defined base for x.
$\circ$: a relational operator from $\{<, \leq, =, \neq, \geq, >\}$.
$\mathcal{R}_c^x$: the identifier set for index component c for value x.
$\mathbb{0}$: The empty set of identifiers for the index, i.e., either a bit vector with all bits set to 0 or an empty position list.
$\mathbb{1}$: The complete set of identifiers for the index, i.e., either a bit vector with all bits set to 1 or a position list containing with all identifiers in I.

Postcondition:

R: represents the identifier set according to $z \circ x$ for all $z \in I$.

```
function RangeEval-Opt($\circ$, $x$, $\beta$)
    $R \leftarrow \mathbb{1}$
    if $((\circ \in \{<\} \wedge x = 0) \vee (\circ \in \{>\} \wedge x = \prod_{i=1}^{k} \beta_i - 1))$ then return $\mathbb{0}$
    if $(\circ \in \{<, \geq\} \wedge x > 0)$ then
        $x \leftarrow x - 1$
    end if
    $x \leftarrow \langle x_1, \ldots, x_k \rangle_\beta$
    if $(\circ \in \{<, >, \leq, \geq\})$ then
        if $(x_1 < \beta_1 - 1)$ then $R \leftarrow \mathcal{R}_1^{x_1}$
        for $i = 1$ to $k - 1$ do
            if $(x_i < \beta_i - 1)$ then $R \leftarrow R \wedge \mathcal{R}_i^{x_i}$
            if $(x_i \neq 0)$ then $R \leftarrow R \vee \mathcal{R}_i^{x_i - 1}$
        end for
    else
        for $(i = 1$ to $k)$ do
            if $(x_i = 1)$ then $R \leftarrow R \wedge \mathcal{R}_i^1$
            else if $(x_i = \beta_i - 1)$ then $R \leftarrow R \wedge \overline{\mathcal{R}_i^{\beta_i - 2}}$
            else $R \leftarrow R \wedge (\mathcal{R}_i^{x_i} \oplus \mathcal{R}_i^{x_i - 1})$
        end for
    end if
    if $(\circ \in \{>, \geq, \neq\})$ then $R \leftarrow \overline{R}$
    return $R$
end function
```

Algorithm A.1: RangeEval-Opt [39] adapted according to our notation. Figure A.1 illustrates how the algorithm operates for a specific value.