WHITEPAPER

Scalyr Under The Hood: Log Management That's Fast, At Scale

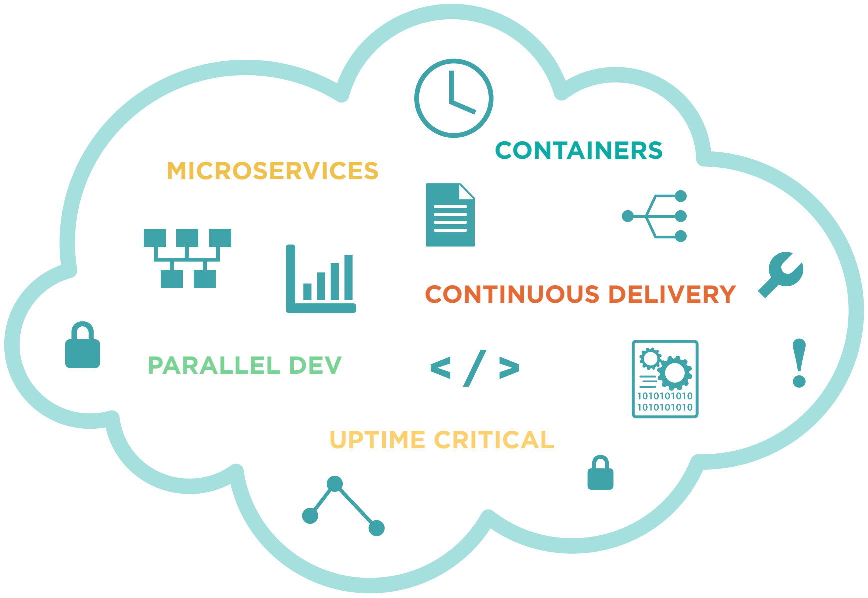
The Engineering Front Line

Your engineering team is on the hook like never before

When the software you build is customer facing and generates revenue, you're not just responsible for its development, deployment, and successful ongoing operation. The work you and your team do is the lifeblood of your business. All eyes are on your team, and the stakes have never been higher.

That's not all that has changed. Your team also needs to move quickly to ensure that the software runs smoothly, and that means they're constantly checking on its performance. Outages, downtime, and performance issues can lead directly to loss of revenue or even be an existential threat to your business.

When it comes to new tools and innovations, you'll find the engineering team at the forefront of your company. You and your team use cloud infrastructure and tools, you may deliver your applications as microservices, you build and consume APIs, and you use container technologies. You may also run analytics, artificial intelligence, machine learning, and even consume sensor data. All of this means that you're dealing with more complexity and more data than ever before.



“When it comes to new tools and innovations, you'll find the engineering team at the forefront of your company.”

Observability is everything

When you're on the front line, operational visibility is critical. The engineers responsible for customer-facing applications rely on logs to get that visibility. To be useful, these tools need to facilitate the way you work today.

Log management is foundational to operational visibility. While APM and metrics tools help engineers understand when there's a problem brewing, only logs provide the full detail required for troubleshooting and understanding the root cause of issues across infrastructure, containers, applications, and APIs.

Traditional log management tools are slow

There's a problem with traditional log management tools. Whether they're delivered in the cloud or on premises, these tools were built for IT cost centers, not the engineering front line. They are slow and do not satisfy the use cases that are most important to you and your team.

When your team encounters a performance issue, they need to move fast. But they don't always know what they're looking for. They need to drill quickly into the log data and perform queries, often in rapid succession. They need to pivot data, apply filters, and create visualizations such as distribution of service latencies, a graph of error events by client type, or spikes in load time for a web server. Traditional log management tools weren't built for rapid, ad hoc querying and they can be frustratingly slow—especially if your team is dealing with large data sets, complex searches, or a large window of time. Users of traditional log management tools find this to be a challenge—one that discourages them from being proactive when it comes to troubleshooting issues. They know that executing a query or visualization across their large data set will come with a stiff performance penalty. Because they can't afford to get bogged down, they often choose not to run the query in the first place. This means your team is only troubleshooting when there is an emergency, and not being proactive by investigating small issues before they become emergencies.

“*Traditional log management tools weren't built for rapid, ad hoc querying and they can be frustratingly slow—especially if your team is dealing with large data sets, complex searches, or a large window of time.*”

Keyword index limitations

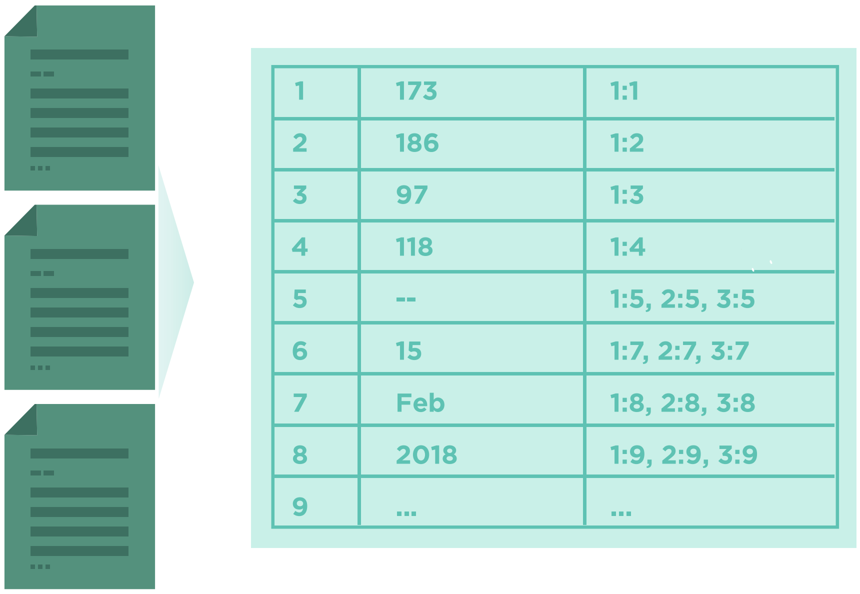
Any kind of log exploration generally starts with a search. The user needs to find all of the messages that match a certain pattern. This search might involve hundreds of gigabytes or even many terabytes of logs collected from many sources.

Traditionally, to search a large data set, you'd use a keyword index. Apply this logic to server and application logs, and that means finding each unique word that appears in the log. For each word, you'd make a list of all log messages containing it. This makes it easy to find all messages containing a single word, like “error” or “Firefox” or “transaction_16851951.” You'd just check the list for that word.

Keyword indexes are great, but they have limitations. Searching for a single word is easy. Searching for multiple words—e.g., messages containing both “Googlebot” and “404”—isn't much harder. Searching for a phrase, like “uncaught exception,” gets a bit trickier, requiring a slightly bulkier index that not only tracks which messages contain a word, but also where in the message the word appears.

The real difficulty arises when your search isn't about words. Suppose you want to see how much of your web traffic comes from bots. As a quick hack, you might search your access logs for "bot." This won't catch everything, but it will match "Googlebot," "Bingbot," and plenty of others. However, "bot," in this context, isn't a word. It's a word fragment. If you look up the word "bot" in a keyword index, you won't find messages with the word "Googlebot."

Punctuation is another challenge. Want to find all requests from 50.168.29.7? Or logs that contain "[error]?" Keyword indexes usually omit punctuation. Sometimes, nothing will do but a regular expression. Using a keyword index for regular expressions is difficult at best.



1	173	1:1
2	186	1:2
3	97	1:3
4	118	1:4
5	--	1:5, 2:5, 3:5
6	15	1:7, 2:7, 3:7
7	Feb	1:8, 2:8, 3:8
8	2018	1:9, 2:9, 3:9
9	...	...

Capabilities aside, keyword indexes are complex and require a lot of storage. Each message has to be added to multiple keyword lists. These lists must be constantly collated and maintained, in seek-friendly form, on disk. Queries that involve phrases, word fragments, or regular expressions have to be translated into operations on multiple keyword lists, and the resulting lists need to be scanned and merged to yield a result set. This complexity, in the context of a large-scale multi-tenant service, can create performance problem and be costly from a storage standpoint.

"Any kind of log exploration generally starts with a search. The user needs to find all of the messages that match a certain pattern. This might involve hundreds of gigabytes or even many terabytes of logs collected from many sources."

Scalyr's Architecture: A "Brute-Force" Approach

To optimize for the way your team works, Scalyr took a different tack. Rather than settle for the traditional search paradigm, we explored a "brute-force" approach consisting of a linear scan through a log file. It worked well, and we were soon convinced that we could provide a rich, exploratory experience at speeds that put competing products to shame. But we needed to do it across very large data sets in the real world.

For this, we created a purpose-built, streamlined database that would ingest log data and store the raw content, with each parsed element in its own column. Simple searches scan all of the log data, byte by byte, at high speed. Targeted queries only need to scan the relevant columns.

Why do it this way? From an abstract algorithmic perspective, keyword indexes can be far more efficient than brute search. However, performance is not merely an algorithms question; it's a systems engineering question. We have to consider everything: How much data are being searched, what types of searches users perform, the available hardware, and the software context in which the search will occur. For the problems engineers face today, a grep-style linear scan is efficient and fast, while avoiding the ingestion cost, complexity, and inconsistent performance of keyword indexes.

"Scalyr took the opposite tack in architecting for the kind of searching you and your teams do. Rather than try to fit the traditional paradigm, we explored a "brute-force" approach consisting of a linear scan through a log file."

Modern processors are blazingly fast when it comes to simple, straight-line operations. It's easy to lose track of that when we're surrounded by the complex, deeply-layered, I/O- and network-dependent systems that are so common nowadays. We put together a system design that minimizes layering and cruft and optimizes for efficiency across large data sets.

The “force” part

We’ve discussed how log search can be implemented as a “brute” problem. Now, how much “force” can we harness? Quite a lot, it turns out. Here’s an overview of how the system has scaled.

One core: A single modern CPU core, properly used, is quite powerful on its own.

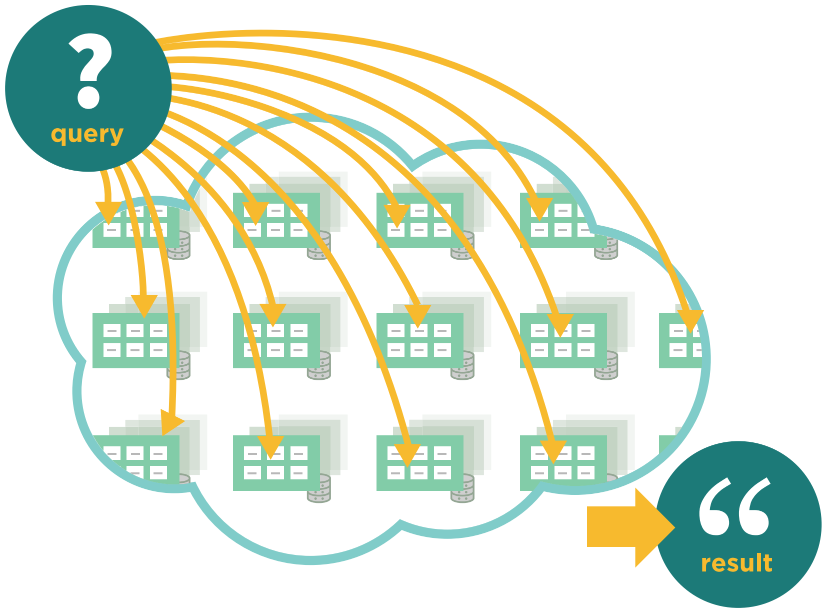
Eight cores: We’re currently using Amazon i3.4xlarge SSD-based servers, each of which has eight cores (16 with hyperthreading). Normally, these cores are busy handling background operations. When a user performs a search, we pause all background operations, freeing all eight cores for the search. The search usually completes in a fraction of a second, after which background work resumes. A governor ensures that a flurry of searches won’t starve important background work.

16 cores: For reliability, we organize our servers into primary/secondary groups. Each primary server has one SSD-based secondary and one EBS-based tertiary. If a primary server fails, the SSD-based secondary can immediately take its place. Almost all of the time, the primary and secondary are both healthy, meaning that each data block is available for searching on two different servers. (The EBS replica has minimal CPU, so we don’t consider it here.) We assign half of each search to the secondary servers, meaning that we have a total of 16 CPU cores to harness.

Many cores: After years of scaling and optimizing, we have spread data across servers in such a way that all of our servers can participate in every non-trivial query. In this way, every core we own comes into play. When combined with high per-core search performance, we have been able to aggregate search performance beyond 1.5 terabytes per second (and growing!).

A simple, brute-force solution means consistent performance, too. This approach tends to not be overly sensitive to the details of the task and data set. After all, they do call it “brute.”

“A simple, brute-force solution means consistent performance, too. This approach tends to not be overly sensitive to the details of the task and data set. After all, they do call it “brute.””



Predictable performance over hit-or-miss-speeds

Our approach means we can move fast under any search conditions. A keyword index can deliver remarkably fast results in some cases, but might not in others. Suppose you have 50 gigabytes of logs, in which the term “customer_5987235982” appears exactly three times. A search for “customer_5987235982” would read the locations of the three matches directly from the index for that term and complete instantaneously. But a complex wildcard search might scan thousands of keywords, which would take a long time to complete.

Brute-force search, on the other hand, will run at more or less the same speed for any query. Long search terms perform better, but even searching for a single character is reasonably fast.

Algorithmic complexity aside, the simplicity of brute-force search means that observed performance will come closer to theoretical performance. There’s less scope for unanticipated disk thrashing, lock contention, uncached pointer-chasing, and all the thousand natural shocks that code is heir to.

See for yourself

When ingesting data and performing queries across large data sets, it’s important to choose a good algorithm. But “good” doesn’t always mean “fancy.” At Scalyr, we know simpler algorithms are easier to optimize. They’re less vulnerable to bad edge-case behavior, as well. That’s why we chose the brute-force approach for the kinds of use cases engineers on the front line face.

Using a brute-force approach, we’ve been able to implement blazing fast, simple, shareable log management for engineers on the front line of software delivery.