Best Practices for Building & Running a Security Data Lake on Amazon S3

Contributed by: Jonathan Rau - VP/Distinguished Engineer, Query

Table of Contents

Introduction to Amazon S3

For almost as long as Hadoop Distributed File System (HDFS) could mount S3 buckets, data lakes (then simply called data warehouses) were built on Amazon S3.

Though you could argue the phenomenon stretches even further back, with S3 a popular durable storage location for raw and archival data for big data and security teams for nearly two decades. As the big data engineering ecosystem kept growing, building data lakes (and later, data lakehouses) on Amazon S3 and other public cloud object storage became table stakes.

With the advent of popular query engines such as PrestoSQL and Trino, as well as the further development of metadata catalogs and metastores such as Hive Metastore (HMS), this option made more and more sense.

To provide potential customers with seamless onboarding, Amazon developed several complimentary services over the years such as Amazon Elastic MapReduce (EMR), EMR Serverless, AWS Glue, Amazon Athena, and AWS LakeFormation.

All of these services are typical cornerstones of building a successful data lake or data lakehouse on Amazon S3.

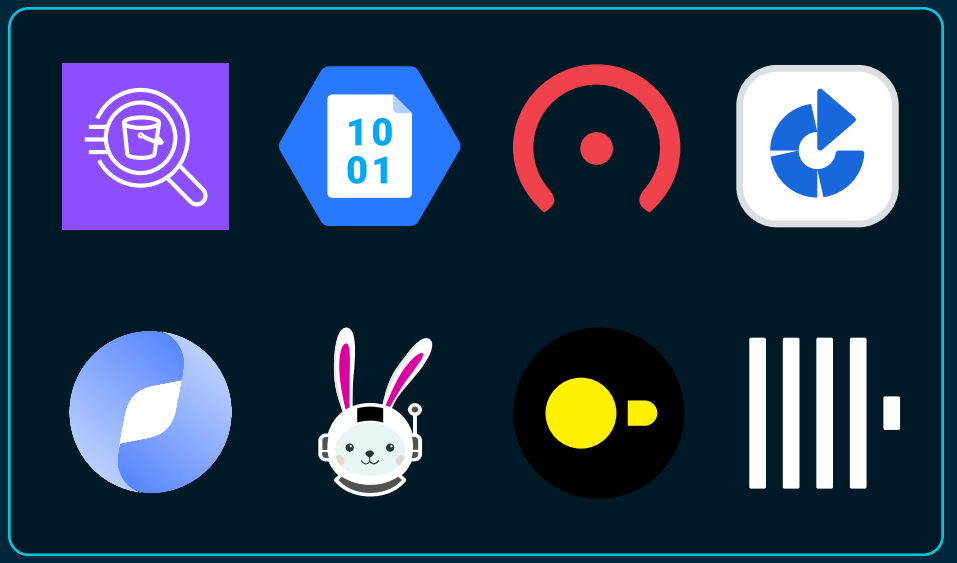
While, in theory, it's simple to move or write bulk data with streaming, micro-batching, or ad-hoc batch workloads onto Amazon S3, catalog it with AWS Glue, and query it with Amazon Athena, there is way more to it than that.

With security teams starting to adopt security data lakes and security data lakehouses and using Security Data Pipeline Platforms (SDPP) and other data mobility technologies, the stakes are even higher for security teams to get it right the first time.

In this whitepaper, you will learn the various features and AWS-native tooling that can be used to build a security data lake or security data lakehouse. You will learn best practices around writing data such as dealing with the “small file problem”, data formats, compression, partitioning, indexing, and open table formats.

Additionally you will learn about efficient query patterns such as optimizing your queries, joins, aggregations, ordering - we will demystify their inner workings. By the end of this paper, you will have the knowledge to fine-tune your security data lake or security data lakehouse performance.

Author’s Note: While these optimization tips are mostly focused on Amazon S3 data being queried by Amazon Athena, they can be broadly applicable to building security data lakes or security data lakehouses on Azure Blob (ALDSv2), other S3 compatible storage (Ceph, Bamboo, Yandex), and querying with a variety of other engines such as self-hosted Trino, DuckDB, or ClickHouse.



You have options while building your securtiy data lakehouse.

WTF Is a Security Data Lake(House)?

Author's Note: If you are already familiar with what a security data lake or lakehouse is, and the common components used within, feel free to skip over the next two sections.

A security data lake and a security data lakehouse are related but distinct architectures used by modern Security Operations (SecOps) and Security Data Operations (SecDataOps) teams to manage and analyze large volumes of security telemetry.

What makes a data lake a data lake is the ability to store disparate data of different kinds in one area and analyze or search it with a separate compute layer. By separating the compute layer from the storage layer, these systems are typically more cost-effective and higher-scaling than traditional databases, OLAP or otherwise (OLTP), and are often far more cost-effective than a traditional Security Information & Event Management (SIEM) tool.

In the simplest terms, a security data lake is a security-focused data lake which is a data repository that can store unstructured, semi-structured, and structured data used for analytics, detections, searching, and visualization. The data is (typically) stored in its raw or native forms, such as in JSON, CSV, TSV, or text files and utilize the relatively cheaper storage of data to contain high volumes of data from EDR telemetry, SaaS/cloud audit and authentication logs, IDS/IPS, firewalls, and similarly voluminous data sources.

Security data lakes built on Amazon S3 utilize the cheaper storage ($23/TB for standard tier) smartly alongside data lifecycle rules to systematically move data of a certain age or type through less-durable (but cheaper) storage tiers before finally archiving them in Amazon S3 Glacier or completely deleting the data sets.

Additionally, specific encryption and access controls can be applied on the data to maintain a higher level of information security and information assurance.

Querying this data has often relied on moving the archival data out of the lake into searchable systems such as Amazon OpenSearch Service or another dedicated SIEM or commercial data intelligence platforms such as Databricks or Snowflake. Data can also be queried directly from Amazon S3 using Amazon Athena, a serverless query engine based on PrestoSQL & TrinoDB, when cataloged or crawled using AWS Glue.

A security data lakehouse builds on the data lake by providing features typically available to data warehouses, also known as Online Analytics Processing (OLAP) (or simply, analytics) databases. Features include the ability to ensure ACID transactions (Atomicity, Consistency, Integrity, Durability) so that reads and writes do not conflict with each other.

Additionally, other features such as schema enforcement, fine-grained access controls, and even advanced features such as liquid clustering, bloom filters, indexing, partitioning, and schema evolution can be supported.

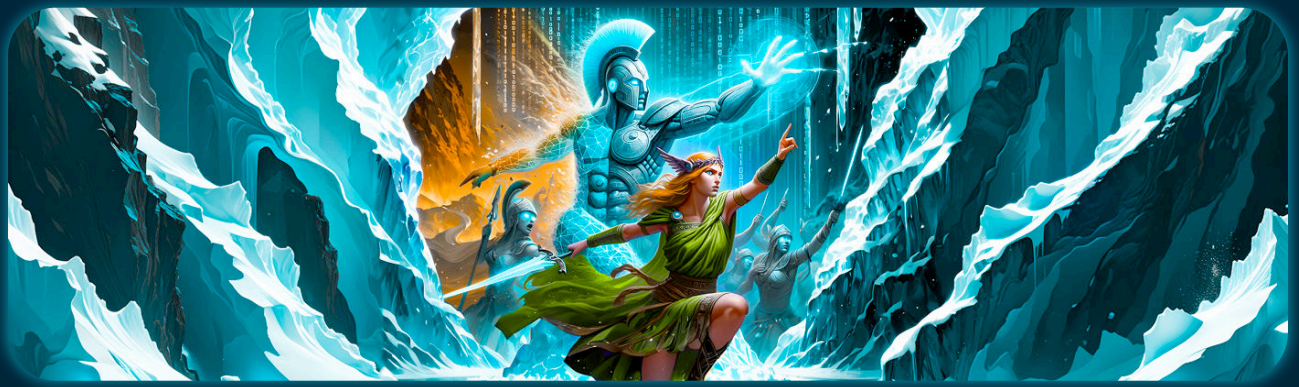
In the AWS ecosystem, the services used to build a security data lake and a security data lakehouse are often the exact same.

The ACID transactions and other features are applied by advanced features of the same services and the usage of open table formats such as Apache Iceberg, Apache Hudi, and Delta Lake. We have written about building Apache Iceberg data lakehouses on S3 here, and have started a series on doing the same with Delta Lake here.

The outcomes are the ability to accelerate migrations away from SIEMs or log management tools to ultimately reduce or at least control costs and to provide more utility with the data.

While mainstream SIEMs do provide a ton of utility in the form of detection content, visualizations, faster index-based search, and automation - not all of these features are required, or desired, depending on the vendor.

Really a security data lake or security data lakehouse is about controlling your own fate, and it's a popular choice for these reasons.



Blog: Amazon Athena and Apache Iceberg for Your SecDataOps Journey

Blog: Delta Lake for Security Teams: Scalable Log Management & Analysis

Common Security Data Lake(House) Components

Now that you know what a security data lake and security data lakehouse is, let's go over the how.

This is a breakdown of AWS services commonly used to build security data lakes and lakehouses, along with examples of external tools (e.g., SDM platforms, orchestrators) that integrate with or enhance these architectures.

Understanding the services used goes beyond the simple "lego blocks" and also gets into the financial due diligence you will be expected to perform as part of your SecDataOps program.

You should thoroughly understand the pricing calculations, and thus understand what cost levers you can pull. While S3 has a simple per-GB storage calculation, it's a little less clear how to forecast out the Get/List API calls that you are also charged for.

Likewise, the scan charges for Athena are easy to understand, but the compute consumption side (DCUs) may not be if you use provisioned capacity.

Common Services

These are the baseline services that are required to have a self-contained system for both the storage and compute layers.

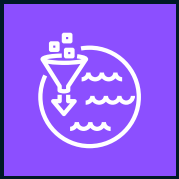
Amazon S3: S3 is an object-storage service that can handle petabytes (or even exabytes) of data. Comes with built in encryption, data lifecycle management, metadata management, and access controls (S3 bucket policies).

AWS Glue: The Glue Data Catalog is a metastore that registers the schema of data stored within specific paths in S3. Advanced users can also register partitions and indexes to optimize query performance. AWS Glue `also provides ETL and data discovery capabilities via Glue Jobs and the Glue Crawler, respectively.

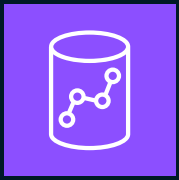
Amazon Athena: A serverless SQL query engine based on PrestoSQL and Trino. When the data schemas are registered in Glue Data Catalog, seamlessly query the data with Athena as if they were native database tables.

Specialized Services

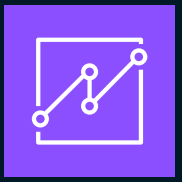
These are services that can be used in addition to the previous three to provide extra access control, governance, querying, and visualization capabilities as well as data streaming and loading.



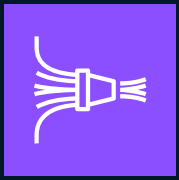
AWS LakeFormation: Provide fine-grained access control to tables, views, or specific columns within Athena/Glue tables backed on AWS. This allows you to control what specific AWS identities can access what data and how. For instance, you can lockdown an AWS IAM Role that has full Glue and S3 access to only be able to execute READ actions on a specific table.



Amazon Redshift Spectrum: Amazon Redshift is a purpose-built OLAP data warehouse used for petabyte scale slicing-and-dicing and analytical workloads. Redshift Spectrum allows you to directly query Athena/Glue tables within the Redshift engine to provide OLAP capabilities without moving all of the data.



Amazon QuickSight: A serverless Business Intelligence and reporting service that can integrate with Athena and Redshift to create visualizations, charts, and apply AI-powered insights using Amazon Q for Redshift.

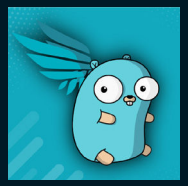


Amazon Data Firehose: Formerly known as Kinesis Data Firehose, this service allows you to perform in-transit ETL and directly write to Amazon S3 tables with specific partition paths, compression, buffering, and data formats. While it can be a relatively expensive service, it is completely serverless and popular for moving native AWS sources into S3.

External Tooling

For data sources outside of the AWS ecosystem, or for host-based data sources such as Windows Event Logs or Kubernetes logs, commercial and open-source tools are a popular choice. Many of these tools, and security-specific pipeline tools called Security Data Management (SDM) tools natively integrate with Amazon S3 similarly to Data Firehose.

FluentBit: FluentBit is a telemetry agent that can be configured to move dozens of discrete types of data from Linux and Windows hosts as well as containerized and Kubernetes environments into Amazon S3. Basic filtering and pre-processing can also be applied.

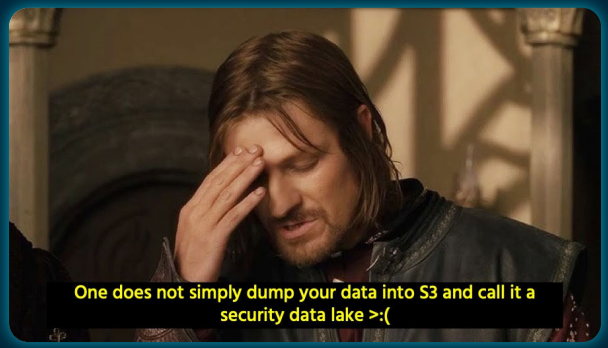


Falco Sidekick: Sysdig Falco is a runtime security agent that uses eBPF and kernel monitoring on Linux, Docker, and Kubernetes environments to detect suspicious and malicious behavior. It is a popular open source monitoring tool, Sidekick is a daemon that can move this specialized data onto Amazon S3 (and several other sources) natively.

Cribl: Cribl is a data telemetry management system that provides a myriad of mechanisms to move host-based data (via Cribl Edge) and other push & pull data sources (via Cribl Stream) to downstream sources such as Amazon S3. Cribl is more synonymous with a Platform-as-a-Service (PaaS) tool that provides a great deal of customization and capability for processing and writing data to S3, but requires deep domain understanding to utilize effectively for that purpose.

Performance Optimization Best Practices

While it is simple enough to create a S3 bucket, dump JSON data into it, torture yourself crawling it with AWS Glue, and querying it with Amazon Athena–that does not make an effective security data lake or security data lakehouse.



And while you certainly could do that and be fine if you only had a few 100MBs or dozens of GBs of data to query in an emergency, it would not be performant at all.

Think of the security jobs to be done that you want to support with this architecture. Is saving a few $1000 a month on SIEM costs worth it when you'll have 10X slower queries that cost almost the same in Athena?

Is it worth being able to immediately join contextual datasets with detections if it takes 2X as long and 5X as expensive versus you analysts opening up 10 tabs in Google Chrome?

What's that tired old mantra? *Don't let perfect be the enemy of good?* Think of the inverse, don't let "good enough" be the enemy of good.

SecOps and SecDataOps teams, as well as all of your stakeholders and customers certainly deserve better and that means meeting a minimum standard of at least having performance optimization versus your legacy SIEM, if not an outright cost optimization to go along with it.

And even if you're a SMB - as an overwhelming majority of companies are with less than 500 employees - don't think that your data volumes cannot grow to significant scales. Even a small startup can easily rack up 10s if not 100s of TBs per month across your entire estate of cloud logs, host-based logs, and other security-relevant signals from productivity and developer tools like Zoom, Slack, GitHub, Hubspot, and more.

So with all that in mind, before we get into optimizing your writes and reads, ensure that you are building a security data lake or security data lakehouse for the right reasons! You should always work backwards from your main stakeholders and SecDataOps requirements.

Sure, you can save oodles and oodles of currency by reducing your SIEM ingestion license and using S3's $23/TB storage, but what happens when you are scanning 100s of GBs to TBs to fulfill your detections?

Amazon Athena is $5/TB of data scanned, simple napkin math shows that a detection across network data such as ZScaler ZIA, Akamai firewall logs, AWS VPC Flow Logs, and otherwise to look for known malicious C2-associated IP addresses or data transfer volumes over specific thresholds can be several times that per-terabyte cost.

Even for the smallest environment, you can rapidly generate several TBs per month (or faster). Running scheduled Athena queries that search these data sources and compare the source IPs to a large static list of data or performs aggregations on "bytes in" that scans just 5TBs of data collectively is already $2 more expensive than it costs to just store a fraction of that data.

The above example is entirely notional but not at all removed from the reality of the types of workloads and queries ran on a security data lake or security data lakehouse.

The first step you must take before starting to build a security data lake or security data lakehouse is asking yourself: *"Do I really need this?"* Here are some questions you may find yourself asking yourself and your team:

- Do we have the appropriate human capital to dedicate towards this project? Have we trained our staff to operate with this architecture? Does anyone know SQL?!
- Can we effectively execute administrative and operational tasks with this new architecture? Do we have robust alerting, monitoring, and the people to do something about it?
- Have we taken a hard look at "worst case" compute costs? Will they counterbalance the cost savings?
- Is the data relevant to our analytical, contextual, detection, and/or enrichment use cases?
- Can we keep the data in place (e.g., in your SIEM, in a traditional database, behind the API) or will we utilize it better in the lake or lakehouse?
- Are you writing too much, or too little, relevant data in your lake or lakehouse?
- Are the reports and advanced use cases we want to support with this data going to make a tangible difference?
- Are our detection use cases tuned effectively? Do they run too often? Do they request too much irrelevant data?

You may find that you are better off staying on the SIEM and reducing the total data size there by offboarding unneeded sources, or removing unnecessary keys or columns. You may realize that you don't have the administrative, financial, nor operational prowess to support moving to a security data lake or security data lakehouse. You may actually find that you don't reduce costs, but maybe you improve operational efficiency, and it could be worth it. Heck, maybe this paper is the key to unlocking operational efficiency?!

All that out of the way, effective cost optimization all flows from the data, so firstly, let's focus on effective writing.

Performance Enhancing Writes

While Performance Enhancing Drugs (PEDs) may give you an unfair advantage in baseball or mixed martial arts, Performance Enhancing Writes (PEWs) are a necessity to get any sort of advantage over a SIEM or other log management system with a security data lake or security data lakehouse.

When all of these best practices are combined they can provide better value-per-dollar by effectively utilizing right-sizing and leading to faster reads downstream, minimizing query runtimes and data scan sizes. Additionally, these best practices should also be deciding criteria from any Security Data Management (SDM) or other data mobility solutions you may look at.

While there are a myriad of ETL enhancements and optimizations to make, if you cannot effectively write the data with those tools, you may be better off going with more customizable options or adopting another layer of ETL in between a "staging lake" and your operationalized security data lake or security data lakehouse.

Minimum necessary fields

As I touched on in the opening to this section, you *really* need to work backwards from your use cases. It is very tempting to keep all of the fields for all of the logs, but utilize previous detection, triage, and response activities and agonize over the details (within reason) to only bring back relevant data that is either necessary for your detection logic or for attribution and triage context.

This should be both a combination of pruning down the fields that you *actually* need and the activities you actually care about. Do you need to have five different timestamps if you are writing detections that implicate when an event happened instead of ended? Do you need the packets transferred if you're just aggregating on bytes? Do you actually need traffic that was allowed? Only you can decide for yourself.

For instance, let's consider a basic Amazon VPC flow log which logs virtualized OSI Layer 4 traffic (TCP/UDP) that comes across your VPCs and hits the various Elastic Network Interfaces (ENIs) attached to it. This is not representative of *all potential fields*, as denoted **here**, you should be even more aggressive when it comes to adding extra data into the log not available otherwise.

This log shows accepted traffic from one source to another, both are internal to the VPC, there are three timestamps as well as the flow log version (the first field, 2).

```
2 123456789010 eni-1235b8ca123456789 172.31.16.139 172.31.16.21 20641 22 6
20 4249 1418530010 1418530070 ACCEPT OK
```

You may consider getting rid of the version (unless you maintain a varied amount), as well as stripping the two "extra" timestamps for start and end of the flow. Additionally, you may completely want to exclude this data since it was accepted traffic and it is also internal data. The ENI ID is the main identifier, but do your triage and IR analysts actually use this for enrichment or pivoting? Is the source IP good enough?

Again, only you can answer this question. The less rows and less columns you carry, the more cost effective you will be in addition to less data that an analyst has to scan through. Not every analyst may know every single data point for every log format (unless you used a standardized data model, more on that later).

Optimal Data Formats & Data Types

Traditionally, security appliances and default log formats were typically written to text files or CSV (or, God forbid, XML 🤮). As RESTful services became more prevalent, logs could be retrieved via API, often in JSON but also those other formats. Some security tools even offer direct exports such as Broadcom Carbon Black or AWS GuardDuty able to support seamless writing into S3.

A commonality with all of these data sources - CSV, text, TSV, XML, JSON, etc. - is that they are horribly inefficient to query at scale. In the case of Amazon Athena, it does support all of these with native Serialization/Deserialization (SerDe) rules but this comes at additional cost due to the extra compute and time it takes to fulfill the query. While analysts can easily open this raw data up and use CLI or IDE tools to read them, at TB scale let alone PB scale this simply will not - well - scale.

You should provide easy to use interfaces to your analysts, that is your frontline infantry afterall, who must be empowered with logistics. The data should use binary, columnar formats that are far more efficient to read, filter, and sort by Amazon Athena and other tools (DuckDB, ClickHouse, et al). King among the binary file formats is Apache Parquet, it is also used by every single major open table format such as Delta Lake, Hudi, and Iceberg.

Besides being very query-efficient, engines that read and write Parquet benefit from increased data types. It's very tempting to turn everything into a string, but unlike JSON, Parquet natively supports datetime types as well as a variety of integer and float types. In a CSV or text file, that would be a string, but attempting to generate a JSON file with a script and writing a Python dictionary with a natively-typed datetime would throw an error.

It's more preferential to avoid the usage of casting or transform operators at query time, such as transforming a stringified `bytes_in` column back into an integer before running a mathematical operation on it.

Many commercial SDM and data mobility tools, such as Cribl, offer the ability to write data as Parquet natively. In fact, many AWS services such as VPC Flow Logs and CloudFront Access Logs also output in Parquet format natively as well.

If you are utilizing a homegrown SDM or ETL pipeline tool, strongly consider transforming the data and writing it as Parquet. You can add tools such as Polars, PySpark, or DuckDB into your homegrown SDM/ETL stacks to write the data to Parquet and retain those proper data types.

Partitioning the Data

A data partition is just like a partition on a disk, or a partition wall you installed in your home office, it keeps specific things in their place. Partitions in Glue are essentially the subdirectories (paths) in Amazon S3 where data matching a specific categorization is kept. Partitions should be built from low low-cardinality columns in your data such as time-based columns (year, month, hour) or on high-level categories such as a specific vendor, source (e.g., firewall, EDR, IDP), or otherwise.

You don't necessarily need to have the partition columns defined in your data source to write partitions in S3, to S3 they're just another folder/path anyway. However, certain open table formats (again, more on that later) do require you to have the columns available in the dataset because of the non-reliance on a data catalog (Delta Lake) or because the partitions are "hidden" (Apache Iceberg).

Before moving any further, what is the *so what*? Partition data written to AWS Glue Data Catalog is used within the query planning phase of an Amazon Athena query. If you specify the partitioned columns in your dataset, Athena will transparently request all of the partitions that match your query pattern and only run the query against the files in those S3 folders/paths. This is also known as *partition pruning* or *partition pushdown* and greatly reduces the time it takes to return results when queries are not directed against the whole table, also known as a *full table scan*.

A popular partition format follows along with how partitions are designed with Hadoop and Hive, this is called a "Hive-like" or "Hive-compatible" partition which requires the partitions to be labeled. To make this a bit more obvious, consider the following path (to stay with our VPC Flow Log examples) that a typical flow log is written into S3 from your log configuration.

```
my-datalake-bucket/AWSLogs/account_id/vpcflowlogs/region/year/
month/day/logfiles0001.txt.gzip
```

The S3 folders, or paths–whatever you want to call it, essentially define the partitions but this won't be automatically written into AWS Glue if you crawled it. Nor will it be automatically written into Athena using DDL unless you provided `PARTITION()` data in your DDL. A lot of data mobility tools, commercial and open-source, will write your paths out like this which is an antipattern for fast onboarding and automatic recognition of your partitions. Again, you could define them in DDL, but then you are stuck automating `ALTER TABLE ADD PARTITION` or `MCSK REPAIR TABLE` queries against each table with this pattern.

In a Hive-like partition format, the paths should look something like this example:

```
my-datalake-bucket/AWSLogs/account_id=account_id/
source=vpcflowlogs/region=us-east-1/year=2025/month=04/day=21/
logfiles0001.txt.gzip
```


If you crawled the data in the folder underneath `s3://my-datalake-bucket/AWSLogs/` each of those labeled columns would be automatically added into the table schema definition and all partitions registered into the Glue Data Catalog. That said, when you write your queries, you will still need to specify the columns which pertain to the partitions. Glue is smart enough to get at the partitions to the "right and left" of the partitions you specify.

Consider the following SQL statement where you want to retrieve the timestamp of your flow logs and certain fields within a certain timeframe, in this case, after April 19th 2025.

```
SELECT
    src_ip
    src_port
    dst_ip
    dst_port
    timestamp
FROM security_lakehouse.vpc_flow_logs
WHERE timestamp >= TIMESTAMP '2025-04-19'
```

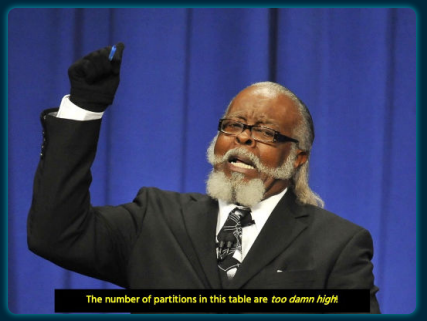
Just because you filtered by `timestamp` in your predicate, that is not enough to actually use the partitions. Now, if you were using Iceberg or Hudi with partitions built off of a specific field, it would, but for the time being assume this is just boring ole' AWS Glue Data Catalog native table format.

To actually have the partitions pruned in your query plan, you would need to specify what specific partitions you wanted.

```
SELECT
    src_ip
    src_port
    dst_ip
    dst_port
    timestamp
FROM security_lakehouse.vpc_flow_logs
WHERE timestamp >= TIMESTAMP '2025-04-19'
AND
    year = 2025
AND
    month = 04
```

In this example, by specifying April 2025 using your predicate aligned to those specific partitioned-by fields, when Athena builds your query plan, it is looking for data with a glob pattern like this: `/AWSLogs/account_id=*/source=*/region=*/year=2025/month=04/day=*/*`. The more partitions you provide, the more precise the pruning will be. Now instead of going through potentially 100s of paths, there will be a smaller amount of data scanned. Now this brings up another potential antipattern: **too many partitions**.

"**Too many partitions?!** What does that mean?" - *You* (😈😈😈)



Look, look, don't hate me, that's what the Athena docs say. In practice, this can be rather complex, but in theory it is not too hard. Remember, the partitions are used for the query plan to only pick up the right data from the right folders/paths in S3. Now, if you had 10s of 1000s of folders, this would still be slowed since the parallelization of Athena is largely an unknown black box - seriously how many paths per "node" are hit? Too many partitions are introduced when you set partitions off of high-cardinality fields such as asset identifiers or attributes such as MAC addresses or IP addresses.

Some vendors do that by default, which is setting you up for failure. (No really, Azure VNET Flow Logs are written with MAC Address AND resource-specific ID partitions AND to the minute-specificity.)

Additionally, if you partition by time too much you have the same issue. An hourly partition is about as far as you want to go, and even then, with properly typed timestamps in your data written in Parquet, you could likely get by with no more than a day partition. Of course, a lot of this is also dictated by the actual queries you write, which we will cover in the next major section around efficiently querying your data. Especially when you start to introduce advanced query patterns such as using `UNIONS` and `JOINS` or building views across two or more datasets, now their partitions also come into scope.

With that in mind, there is a way to solve this though. As your security data lake or security data lakehouse tables and partitions grow in number you can start to implement Indexes over the partitions. Whatever partitioning strategy you choose, be consistent with it so that you can make future advanced query patterns possible without throwing errors and making your SecDataOps team cry out in agony, gnash their teeth, and claw off their clothes.

Author's Note: Consider reading one of my blogs from last year, **Auto-partitioning your Security Data Lake with Apache PySpark and Amazon EMR Serverless**, where I demonstrate how you can use EMR to apply partitions on some truly big data. Additionally, while I did not cover it, you can consider using **partition projection** to avoid needing to redefine tables while also not using `ALTER TABLE ADD PARTION` or `MCSK REPAIR TABLE` statements.

Blog: Auto-partitioning your Security Data Lake with Apache PySpark and Amazon EMR Serverless

Utilize Partition Indexes

As you learned in the previous section, partitions are good. Use them! That's how you keep your data organized and with proper pruning, it leads to much greater performance versus full-table scans over your data. You also learned (hopefully) that eventually, no matter how stringent you are in ensuring minimum necessary partitioning, you will likely accumulate a lot of damn partitions.

To help account for this, as your partitions grow and you notice query performance drop, consider implementing partition indexes. A partition index is a performance optimization feature that helps speed up queries by indexing the partition keys of a table. The index is built from a subset of the partition columns already defined in the table, you can group one or more different partitions into an index which will fetch any/all of the partitions defined in the column at query time.

You can build the indexes off of integer, string, and datetime partitions and support up to three different partition indexes per table. You should build them to fulfill common query patterns, going back to the previous example of the flow log table in the previous subsection, you could build an index that preloaded `account_id`, `year`, `month`, and `day`. This would be helpful if you were constantly querying across data in specific accounts, such as your workloads in specific AWS Account IDs for your Production environment(s).

Indexes will automatically update as new partitions are created, however, after you set the index you cannot (nor should you even want to) modify the partitions. If you add new partitions after the index is created, you'll need to ensure they do not change the order of the previous partitions and you'll have to rebuild your indexes to account for any new partitions.

To define a partition, you build them using the AWS API (via the Console, API, CLI, or SDKs) such as the Amazon SDK for Python, `boto3`. Building on the previous given example, you could build the index like this.

```
aws glue create-partition-index \
  --database-name security_lakehouse \
  --table-name vpc_flow_logs \
  --partition-index '{
    "Keys": ["acccount_id", "year", "month", "day"],
    "IndexName": "account-yyyymmdd-index"
  }'
```

Frustratingly enough, while Redshift Spectrum, Amazon EMR, and AWS Glue ETL Spark DataFrames will automatically work with these partitions but Athena will not, you will need to modify your table to enable partitioning filters with the following SQL DML.

```
ALTER TABLE security_lakehouse.vpc_flow_logs
SET TBLPROPERTIES ('partition_filtering.enabled' = 'true')
```

As always, ensure you are working backwards from the querying use cases, be it detections or ad-hoc queries to design your indexes in an efficient manner.

Avoid the "Small File Problem"

If you've read any of our previous technical blogs on SecDataOps, or have read even the most basic of literature about data lakes and data lakehouses, you probably have heard of the "small file problem". Athena, and other query engines that execute queries on data in object storage have to read out the contents of the actual files, duh. As touched on with partitioning, the larger that query plan and the larger the actual query execution is, the less performant and more expensive your queries will become. Sad!

Another major antipattern you can introduce in your security data lake or security data lakehouse is writing way too many small files, even worse when they are in an inefficient data format and not partitioned. It's much faster for Athena (and other engines) to read one big (200-400MB) Parquet file than it is to read 100 smaller Parquet files that aggregate to the same amount (2-4MB each).

This performance is exponentially faster when actually using Parquet, you will certainly get some performance bump with a similarly large JSON file versus a ton of smaller JSON files, but not as much as Parquet.

The easiest way to avoid this issue is to address it in your pipeline(s), but again, you must work backwards from your detection and searching use cases. If you have a requirement for near real-time data because you have a certain detection that needs to be run at a fast cadence, it may be unacceptable to delay and batch many smaller datasets into a bigger file. This is only something that you can determine.

For a majority of security use cases that will use a security data lake or security data lakehouse, you may need that extremely low latency. Tools such as Amazon Data Firehose or Cribl Stream can buffer data in a certain time and have desired file sizes configured.

However, this needs to be taken into consideration against your partitioning strategy. For larger environments, it wouldn't be difficult to generate a TB if not more of data in a single day, maybe even a single hour. Even when buffering your writes and writing into the optimal size of 128 MB (as per this AWS blog on performance tuning) that still ends up being several 1000 files per TB. When possible, you should fine-tune the sizes of your files against your query patterns by looking at the stats of the planning and execution phases of your queries. You can also use that data to further introduce new partitions and indexes as required.

Other ways to contend with the "small file problem" is post-hoc processing. For instance, running a nightly job that will read the previous 24 hours of data and combine the files into more optimal sizes, especially if you cannot buffer enough data in your necessary latency SLA to get the files to the optimal size. Additionally, you can consider using an open table format such as Iceberg with AWS Glue that can automatically compress and optimize the files. Delta Lake can also make use of the optimized writes and post-hoc optimization using the Delta PySpark extension.

At a certain scale in your security data lake or security data lakehouse, you will have wrung out all of the compaction and optimization that you can get, it will be up to your partitioning, indexing, and query patterns to further optimize your query performance. At the end of the day, you are not only measured on compute and data scan sizes in Athena, but also the S3 APIs like List/Get Object. Having a lot of smaller files can rack those costs up quickly, as well as potentially outright fail your queries due to rate limiting.

Efficient Compression

Finally, we come to compression. If you've worked on a computer for any length of time you are probably familiar with ZIP, RAR, or tarballs that downloads often come compressed with. Compression codecs of various kinds are used to, well, compress the files so they take up less space on disk. This is great for cost optimization of storage, but when you're querying that data you will want to use an efficient compression codec to ensure your query performance does not suffer.

This subsection assumes that you will, rightfully, use Parquet as your data format. Besides being columnar and optimized for selective reading of fields, Parquet files are also splittable and can utilize very efficient compression codecs. Read more here about what file formats and what compression are supported by Athena, we will only focus on three of them.

Really when it comes to compression you have to strike a balance between cost optimization and query performance optimization. That is greatly oversimplified, because the actual queries will cost money as well, so even if you index on storage costs you may end up nullifying that advantage with poor query performance.

The three major compression codecs to use with Parquet data are GZIP, Snappy, and ZSTD. These are all very well supported by streaming tools as well as data engineering libraries such as Pandas, Polars, and PySpark.

Compression Type	Compression Ratio	Decompression Speed	Athena Performance	S3 Cost Efficiency	Notes
SNAPPY	Medium (~1.5–2x)	Very Fast	Best for speed	Less savings	Great for low-latency interactive queries
GZIP	High (~2.5–4x)	Slower	Slower queries	Best storage savings	CPU-intensive; good for archival or infrequent access
ZSTD	Very High (~3–5x)	Fast	Balanced	Good savings	Best of both worlds — especially for large datasets

The compression codec that I personally use for our own security data lakes and security data lakehouses (yeah, we have multiple) at Query is ZSTD. It offers the best of both worlds with even more compression ratio versus GZIP, which is default compression codec used by a lot of security sources that support direct writes into S3 buckets.

While we may use ZSTD, it may not be the best, consider the following table for a summary of compression characteristics.

However, you may find that the cost savings are not worth the additional performance penalty compared to Snappy, which compresses and decompresses incredibly fast — hence the name.

This performance penalty may not become apparent until you are nearing petabyte scale data storage, which shakes out with host-level telemetry from EDRs or from firewall appliances that serve up a high amount of data.

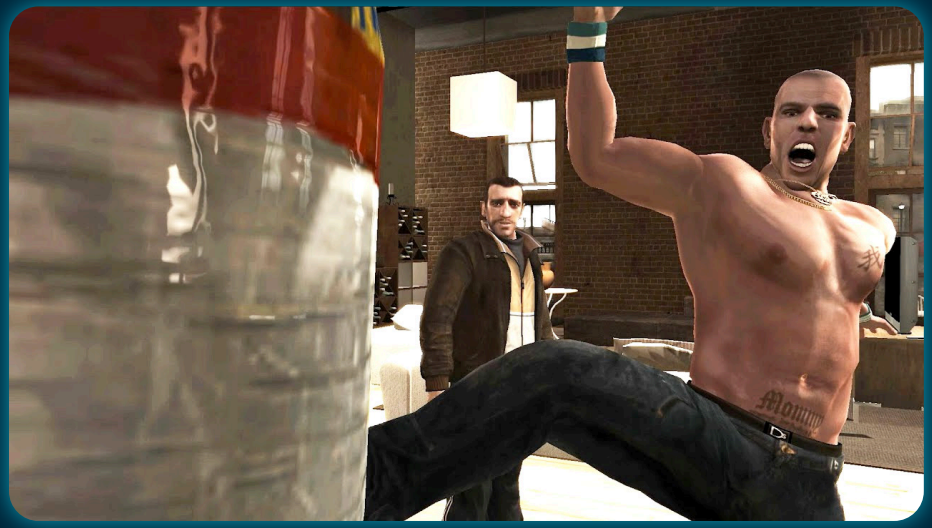
Are you noticing a theme? The Ops part of SecDataOps is incredibly important to inform your choices and possible actions here, as well as the financial implications.

It is worth mentioning that you may completely eschew compression if you have relatively small amounts of data written per typically queryable time periods. That will give you the best overall performance, as even Snappy-compressed data has CPU overhead to decompress and ultimately query.

Metric	SNAPPY	GZIP	ZSTD
Compression Ratio	~1.5–2.0x	~2.5–4.0x	~3.0–5.0x
Compression Speed	Very Fast	Slow	Fast
Decompression Speed	Extremely Fast	Slow	Fast
CPU Usage (Decompress)	Slow	High	Moderate

Performance Enhancing Reads

We're not done with the performance enhancing jokes yet. If optimizing your writes into S3 for your security data lake or security data lakehouse is just a "pictogram" of the "gear", then optimizing how you query your data is like injecting 69420 deciliters of Chilean Bull Shark Testosterone directly into your SecDataOps program. NICO BABY, WE'RE WINNERS!



Author's Note: GTA IV jokes aside, bull shark testosterone isn't a real thing, Chilean or otherwise.

While enhancing the performance of your query patterns may not allow you to defend, and then get stripped of your light heavyweight strap you will certainly feel like a star. Or at least, feel less bad when you look at how long that query plan and query execution took in Athena in preparation to explain to your leadership why that "Big Money Saving SIEM Migration" thing ain't exactly working out...

Actually, nevermind, one more fitness reference. If optimizing your writes into your security data lake or security data lakehouse is a good diet, optimizing reads from them is actually working out. While you can certainly get a lot of performance mileage, so to speak, from optimizing writes it is how you ultimately use the data that will be the real measuring stick for performance.

In this section, we'll dive into some high level query optimizations you can take into account when developing your detection content or search patterns. Obviously, since we're talking about Athena, this will be a very SQL-heavy section.

Limit Fields Queried

This is about as "SQL 101" as it gets here. Remember that whole minimum necessary thing? Well it applies up and down your whole stack. Whether creating detection content or performing ad-hoc searches or writing analytics, you want to only bring back the data you'll need.

Just because you right-sized the amount of columns (or keys) that you'll ultimately define in your tables, doesn't mean that every single query needs to retrieve the same data.

While it's very hard to estimate performance efficiencies in the form of percentages of data scanned or the amount of time that the query ultimately takes, you will always be faster when you request specific data from your Athena tables versus using `SELECT *`.

Again, going with the VPC flow logs table example, grabbing the right amount of data to fulfill your query is paramount.

So do more of this:

```
SELECT
    src_ip
    src_port
    dst_ip
    dst_port
    timestamp
FROM security_lakehouse.vpc_flow_logs
WHERE timestamp >= TIMESTAMP '2025-04-19'
```

And do less of this (actually, don't do this):

```
SELECT
    *
FROM security_lakehouse.vpc_flow_logs
LIMIT 1000000
```

Even in Parquet, reading unnecessary columns increases data scanned and decompression overhead. By selecting only the fields that you need you will reduce memory usage, speed up execution, and lower query cost with Athena.

Partition Pushdown

This is largely redundant with what was detailed in the Partitioning the data subsection, but it is worth repeating again.

Your partitions are not automatically applied.

*(**Author's Note:** they can be in certain instances with Hudi and Iceberg!)*

When you're using the default Glue table format, you must always specify the partitioned field(s) in your predicates. This goes doubly for tables that use partition indexes, you won't get any of the performance benefits of partitioning nor indexing if you are not specifying the fields.

While using the Athena console in AWS, the fields that are partitioned will be denoted with a (partitioned) label next to them, this information is gleaned from the Glue Data Catalog. If you will be remotely querying with Athena using the SDK or another intermediary service, it would behoove you as part of your SecDataOps governance duties to document all of the table schemas along with partitions and indexes. Ideally, in a rich-text searchable back end such as Confluence or another tool, and not just living in a Slack message.

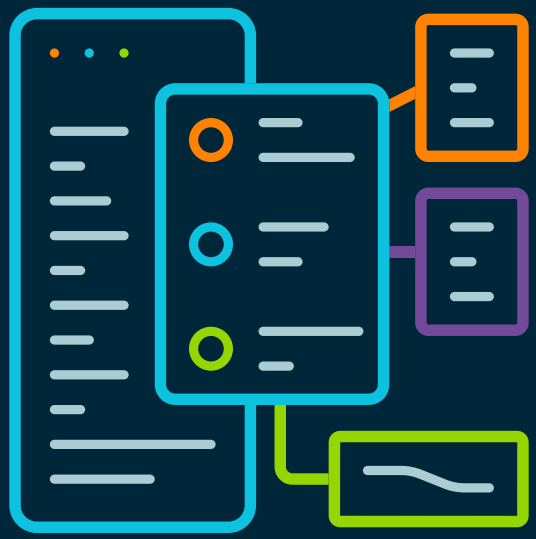
Information access is a big part of a successful adoption for a SecDataOps program, as well as your security data lake or security data lakehouse journey. You could apply that diligence to just about everywhere - your pipelines, your data sources, normalization schemes, table metadata, and more!

Optimized Ordering

The SQL `ORDER BY` clause will order a result set, ascending or descending, off of a specific column. These types of queries are helpful for determining "Top N" or "Bottom N" insights from your various tables.

You should almost always use these with a `LIMIT` confined by how many insights you're trying to find. If the data is going into a visualization or report, think hard about the readability as well as the usefulness of providing "Top N" or "Bottom N" sort of metrics.

```
SELECT
    src_ip,
    bytes_in
    timestamp
ORDER BY bytes_in DESC
LIMIT 100
```



Optimized Grouping

The SQL `GROUP BY` clause is used to create summary aggregations of your data, helpful for measuring averages or other mathematical operations by specific fields in your data. For instance, as an extension of a "Top N" query for a visualization, you can use `GROUP BY` to get the top 25 external IP addresses by data volume, which may be useful for firewall or IDS rule finetuning or as part of determining data exfiltration impacts. For more information on `GROUP BY` and some hands-on examples, check out our **Introductory SQL for SecOps: Exploratory Data Analysis with DuckDB** blog.

When performing aggregations, only include necessary fields in the `GROUP BY` clause to minimize CPU and memory usage within the Athena nodes. Athena distributes rows to worker nodes based on a hash of the `GROUP BY` columns. It's also important to group by columns that have a uniform distribution of values, as this helps balance the workload across nodes. If the data is unevenly distributed, one node may handle a disproportionate amount of the data, leading to performance bottlenecks while other nodes remain underutilized.

Sometimes, redundant columns are added to the `GROUP BY` clause due to SQL requirements, any selected field must be either grouped or aggregated. For example, if you're grouping by `src_ip` and also selecting `src_port`, you may end up writing `GROUP BY src_ip, src_port`, even though `src_port` is uniquely determined by `src_ip` in each row of your flow log data.

To avoid this and improve performance, you can use the `ARBITRARY()` function. It returns an arbitrary value from the group and allows you to select values like `src_port` without including them in the `GROUP BY` clause.

If you will be grouping and ordering, apply the ordering last, and ensure that a `LIMIT` is defined overall that is both actionable and relevant to whatever report or investigation you're conducting.

Optimized Joins

SQL `JOINS` are an advanced function that combines data across two or more tables, useful for enrichment and analytical queries. For instance, you may wish to join your VPC flow log data by `eni_id` into another table (`ec2_asset_info`) to retrieve asset-specific information such as the `hostname` or `instance_id`. This can also be used to join disparate log sources together to tell the full story of the flow of traffic such as joining Amazon CloudFront, AWS WAF, Amazon Application Load Balancer, and VPC flow logs together by a specific source IP. However, doing something that complex can lead to a hefty performance degradation and the query may outright fail.

The reason for this is that, using typical equality-based joins, Athena uses a distributed hash join strategy. It builds an in-memory "lookup table" from the right-hand table and distributes it to all worker nodes. The left-hand table is then streamed, and rows are joined on matches in the lookup table. Because the lookup table is in memory, keeping the right-hand table as small as possible reduces memory usage and speeds up the join. The more tables that are used, the slower the performance will be.

Just like with ordering, data skew can negatively impact join performance. If many rows share the same join key values, a single worker node may receive a disproportionate amount of data to process, leaving other nodes underutilized. To ensure efficient parallelism, try to use join keys with a uniform distribution of values across rows.

For further optimization, ensure that you are filtering first in your `WHERE` clause to select a specific IP address along with other behaviors, and apply your partition pushdowns. Remember, this goes back to the optimized write section, you'll want to ensure that you have consistent partitioning (and indexing) strategies across all of your tables in your security data lake or security data lakehouse. You can also consider selecting distinct values with `SELECT DISTINCT` and even applying aggregations beforehand to further speed up the query execution.

For more advanced uses, you can use `CREATE TABLE AS SELECT` (CTAS) to create an intermediary table of an even smaller size and apply a dynamic partitioning strategy along with dropping empty values for the data that you want to join on such as in the following SQL statement.

```
SELECT
    src_ip
    src_port
    dst_ip
    dst_port
    timestamp
FROM security_lakehouse.vpc_flow_logs
WHERE timestamp >= TIMESTAMP '2025-04-19'
```

For extremely complex joins, you should still consider further breaking up the queries into multiple steps and using intermediary SecDataOps automation workflows to perform the joins in memory by retrieving the filtered data from Athena and then writing the results to Polars DataFrames, Arrow tables, or using DuckDB on the raw files to accomplish the join and not put so much cost pressure on Athena.

Avoid Functions and Casting

Calling back to the **Optimal Data Formats & Data Types** subsection, you should avoid using functions and casting when at all possible. Casting is an operation that dynamically changes the data type of a specific field, or fields. This is common to use when using poorly written tables such as data that stringifies all fields in a JSON file, you'd need to use casting to transform specific fields into specific types to be able to use prebuilt functions.

For instance a statement like `SELECT SUM(CAST(bytes_in as INTEGER)) as sum_bytes FROM security_lakehouse.my_bad_vpc_table ORDER BY sum_bytes DESC LIMIT 1000` would take much longer than executing the same query where `bytes_in` is the proper integer data type.

Athena can only push filters down to the storage layer (like S3) if the filter expression is simple. Using a function or cast on a column blocks predicate pushdown, causing Athena to scan more data than needed, and defeats the partitioning and indexing strategy you use anyway. Like anything else in Athena, operations are applied per row and will consume extra CPU per row processed, so casting can negatively affect performance on very large datasets.

Likewise, casting will negatively affect joins by disrupting the hashed join operation that happens under the surface; it's more expensive and less efficient to hash a field when a cast, a function, or both are used.

Again, this can be prevented by ensuring that your field is in the correct data type for the operations that will be applied against. This goes doubly for more specialized functions such as `FROM_ISO8061_TIMESTAMP` that will convert a stringified ISO 8061 timestamp into a proper SQL `TIMESTAMP` data type.

Not all functions are necessarily "bad" though, using mathematical aggregation such as `COUNT()` and `SUM()` are native aggregate functions that are optimized by Athena. However, if you'll be using the functions in `JOIN` or `WHERE` clauses, the performance penalty will resurface. Again, working backwards from your use cases should drive your table schema design!

Consider Using Views

A SQL view is a logical table built from one or more other tables, this can be useful to simplify complex query patterns by defining a view with all of your aggregations, joins, unions, and other specialized operations.

In cases where you SecDataOps analysts and engineers are directly authoring queries instead of scheduling them or used outside of automated detection content, having views defined can reduce cognitive load and ensure that well optimized queries are predefined.

Views ***will not provide any performance improvements***. Views in Athena are not cached or materialized and essentially work as inlined functions that will apply your query logic at query time. So predefining complex joins in the view is not different than executing a complex join in your SQL statements.

If you are finding that you are making views to disaggregate certain datasets (such as using `CROSS JOIN UNNEST` to loop through arrays) or to cast and convert several data types, this type of feedback must make its way back up your ingestion loop.

Consider Normalization

As we write about in a lot of our content, we love the Open Cybersecurity Schema Framework (OCSF). If this is your first time reading about the OCSF, or if you are coming back to it after an absence, consider reading our beginner and executive-friendly blog: Query Absolute Beginner's Guide to OCSF. For a more detailed explanation of OCSF, see our Definitive Guide to Open Cybersecurity Schema Framework (OCSF) Mapping blog.

OCSF is a hierarchical, strongly typed data model that provides guidance for normalization and standardization of data. OCSF is made up of several Event Classes which are essentially generalizations and representations of common security-relevant data sets such as the Detection Finding event class that can represent alerts or detections from DLP, DSPM, EDR, or CSPM tools. It provides normalized fields for common data types, for instance, recall the complex four-table join across HTTP, ALB, CloudFront, and VPC Flow logs from the previous subsection.

Not only is that a very complex join, all of the default fields in those different tables are all different. By default the "source IP" is represented in the following ways:

- VPC flow logs: `srcaddr`
- CloudFront access logs: `c-ip`
- ALB access logs: `clientIp`
- WAF access logs: `httpRequest.clientIp`

You would be forced to use several intermediary CTAS statements or `UNION ALL` between the different tables to represent the relevant fields as `src_ip`. OCSF helps solve this by accounting for the various permutations of this data. In this same example, all of those fields would be normalized as `src_endpoint.ip` and conversely the "destination IP" as `dst_endpoint.ip`. Having a standardized way to reference common data points in security-relevant data will greatly streamline the development of ETL pipelines, detection content, and visualizations for reporting.

There are several tools that will help you convert data into the OCSF, or, perhaps consider using Query Federated Search. We have a no-code workflow that supports nearly every popular query engine and systems for security data lakes and security data lakehouses: Amazon Athena, ClickHouse Cloud, Google Cloud BigQuery, Amazon Redshift, Snowflake, Databricks, as well as popular SIEMs.

You can keep your optimized written security data lakes or security data lakehouses tables in their native formats, and use our Configure Schema workflow to dynamically normalize that data at query time. If you are interested in seeing a demo of that, hit me up on LinkedIn or reach out to our sales folks for a demo. Operators are standing by.

Wrap Up

As you can (hopefully) tell from this whitepaper, one does not simply build a security data lake, a lot of optimization work must be considered at the forefront.

In this paper, you learned the core building blocks and best practices required to build a high-performance security data lake or lakehouse on Amazon S3. From addressing the small file problem and selecting optimal data formats to implementing partitioning, compression, indexing, and leveraging open table formats, this guide walked through the critical components needed to write and query data efficiently. You also gained insight into performance optimization techniques for common query operations such as joins, aggregations, and ordering, ensuring your architecture can scale with security telemetry and detection needs.

By applying these practices, security teams can fine-tune their data lakes and lakehouses for better cost-efficiency, faster query performance, and higher-quality analytics and detections. Take your next step: evaluate your current setup, start optimizing key areas, and embrace a modern data architecture that enables scalable, flexible, and future-proof security operations.

Until next time...

Stay Dangerous

Query: Making Open Federated Search for Security a Reality

Query aims to deliver visibility into all relevant data for security teams. We provide a **federated search solution** that allows operators to **access data at the source** and in your data lakes, creating opportunities for more nimble and cost efficient data storage architectures.

Our customers are using Query to expand visibility for security investigations, threat hunting, and incident response. They are drastically **reducing the time and complexity** of repetitive search tasks and **improving outcomes for investigations.** Expose your security data with Query.

Ready to expedite your security investigations with open federated search for security?

For more information visit: www.query.ai