ES|QL (ELASTICSEARCH QUERY LANGUAGE)

ES|QL

Revolution in Data Analysis and Computing

elastic

Search. Observe. Protect.

The New Frontier in Data Analysis

Discovering the Power of ES|QL

by **Customer Architect Team**

Meet the Customer Architect (CA) Team, your strategic ally in the Elasticsearch universe. We are here to ensure your success with Elasticsearch and that you get the most out of every feature.

We present you with this essential ES|QL Handbook. This handbook not only facilitates your introduction to ES|QL, but will also help you optimize your data analytics processes. Our goal is clear: to make ES|QL accessible, understandable and valuable to you.

Get ready to transform your analytics approach with expert guidance from the CA Team.

www.elastic.co

To the entire Elasticsearch team: your passion, dedication and knowledge are the backbone of this book.

Thank you for turning every page into a reflection of our collaboration and commitment to excellence.

Together, we continue to build more than solutions, we build possibilities.

Prologue

In a world where data not only abound but flow at dizzying speeds, the need to understand, process, and extract value from them has never been more crucial. Elasticsearch has emerged as a leading solution, offering unprecedented power and flexibility in data management. However, like any powerful tool, unlocking its true potential requires knowledge, skill, and, most importantly, a reliable guide.

This is where this ES|QL Handbook comes into play. Conceived from the collective experience of the Elasticsearch team, this book is not just a compendium of instructions; it's a compass in the vast landscape of data analytics. Through its pages, ES|QL reveals itself not just as a query language but as a bridge to a deeper understanding and more efficient management of your data.

This Handbook is a testament to our dedication not only to technological excellence but also to the success and growth of our users. Each chapter, each example, each tip, is designed with one purpose: to empower those who read it to not only execute queries but also unleash the full potential of their data.

We embark on this journey knowing that every dataset tells a story, every query unlocks a mystery, and every insight could be the start of something extraordinary. May this Handbook be your companion on this journey, illuminating your path, enriching your understanding, and ultimately transforming your relationship with data.

Welcome to a new era of data analytics.
Welcome to the world of ES|QL.

Index

Prologue 5
Introduction to ES|QL 8
Getting Started with ES|QL 13
Use Cases of ES|QL 17
Syntax of ES|QL 22
Commands of ES|QL 25
Functions and Operators of ES|QL 34
Metadata fields of ES|QL 36
Multi-valued fields in ES|QL 38
Processing Data with GROK and DISSECT 39
Enriching Data 42
Using ES|QL 46
Limitations 51
Examples 54
Resources 56

Introduction to ES|QL

At the forefront of query technology in Elasticsearch, ES|QL offers a sophisticated and powerful query interface that transcends conventional data analysis methods. This query language provides users with a more intuitive and direct means to interact with their data, offering unprecedented filtering, analysis, and transformation capabilities. With its user-centric design, ES|QL proves to be an essential tool for those looking to delve deeper into their data and achieve valuable insights efficiently.

Overview

Elasticsearch Query Language, is a powerful tool designed to filter, transform, and analyze data stored in Elasticsearch. This language and computing engine is characterized by its ease of learning and use, making it accessible to a wide range of users, including SRE teams, developers, and administrators.

One of the distinctive features of ES|QL is the use of "pipes" (|) which allows for the manipulation and transformation of data sequentially. This enables the composition of a series of operations where the output of one becomes the input of the next, thus facilitating complex data transformations and detailed analysis.

Moreover, ES|QL goes beyond being merely a query language; it is a new computing engine within Elasticsearch. Unlike other query languages, ES|QL expressions are not compiled into Query DSL for execution but are run directly within Elasticsearch, contributing to its high performance and versatility.

The execution engine of ES|QL has been designed with performance in mind, operating in blocks rather than by row, focusing on vectorization and cache locality, and leveraging specialization and multithreading. This approach ensures that ES|QL is extremely efficient in performing searches, aggregations, and transformation functions.

Fundamentals of ES|QL

Designed to offer an advanced and highly efficient querying experience. This section aims to unravel the fundamental principles of ES|QL, providing a solid foundation for those looking to explore its capabilities.

The Declarative Nature of ES|QL:

ES|QL is a declarative language, which means it focuses on the "what" rather than the "how" of data manipulation. Users specify the desired outcome without detailing the exact steps needed to achieve it, allowing the system to optimize and determine the most efficient process.

Pipe-Based Syntax:

The syntax of ES|QL centers around the use of "pipes" (|), an approach that allows chaining multiple commands to transform and analyze data. This method promotes readability and facilitates the construction of complex queries by breaking them down into sequential, manageable steps.

Query Composition:

ES|QL allows for the composition of queries through a series of commands and functions. Each command processes the data and passes the result to the next command in the chain. This structure facilitates the

performance of complex filtering, aggregation, and data transformation operations.

Deep Integration with Elasticsearch:

Unlike other query languages, ES|QL is deeply integrated with Elasticsearch. Queries are executed directly within Elasticsearch, leveraging its distributed infrastructure and indexing capabilities to deliver fast and accurate results.

Performance and Optimization:

The execution engine of ES|QL is designed for high performance. Operating in blocks and not by row, it focuses on vectorization and cache locality. Additionally, the engine leverages specialization and multithreading to optimize query operations.

Flexibility and Extensibility:

ES|QL not only adapts to a wide variety of use cases but also offers the flexibility needed to extend and customize queries. It supports a range of commands and functions and adapts to different analytical and data processing needs.

The Computational Infrastructure of ES|QL

At the heart of Elasticsearch, the computational infrastructure of ES|QL stands out as a fundamental pillar, providing the robustness necessary to handle and process vast data sets with commendable efficiency. This advanced infrastructure greatly benefits from Elasticsearch's distributed architecture, allowing for efficient scaling and rapid query execution. More than just processing, the ES|QL engine is synonymous with intelligence and adaptability; it is endowed with advanced mechanisms for dynamic query optimization, adjusting operations in real-time to ensure data processing that is not only fast but also incredibly efficient.

But the versatility of ES|QL does not end there. With extensive support for advanced operations, from windowing to time series analysis, ES|QL presents itself as an exceptionally adaptable and powerful tool, suitable for a diversity of analytical scenarios and needs. This versatility is further enriched by its close integration with visualization tools like Kibana, which allows users not only to delve into their data but also to bring insights to life through intuitive visual representations. This holistic approach is reinforced by the constant commitment of the Elasticsearch team to the enhancement and updating of ES|QL, ensuring that this tool not only remains at the forefront in terms of performance and features but also stays in tune with the changing dynamics of the data world.

Getting Started with ES|QL

Starting in the world of ES|QL is an exciting adventure that begins with the proper preparation and setup of the environment. Before diving into the depths of this query language, it is essential to ensure that all necessary components are in place. This includes having an operational instance of Elasticsearch and familiarizing yourself with the interfaces through which ES|QL will be used, such as Kibana or the Elasticsearch REST API. This initial stage is crucial to ensuring a smooth journey in data analysis with ES|QL.

Ready to dive into the world of ES|QL and see what your data can really do? Create your free account at Elasticsearch Cloud at https://ela.st/startnow.

Once the environment is ready, it's time to play, experiment, and discover! The first steps in ES|QL involve understanding its unique syntax, mastering a variety of specific commands, and understanding how to work with functions and operators. Additionally, it is crucial to familiarize yourself with metadata fields and how to handle multi-value fields. Learning to process data with tools like Grok and Dissect, and enriching data are also fundamental parts of this learning. Each aspect is crucial to leveraging the power of ES|QL in Elasticsearch for complex data analysis.

Syntax:

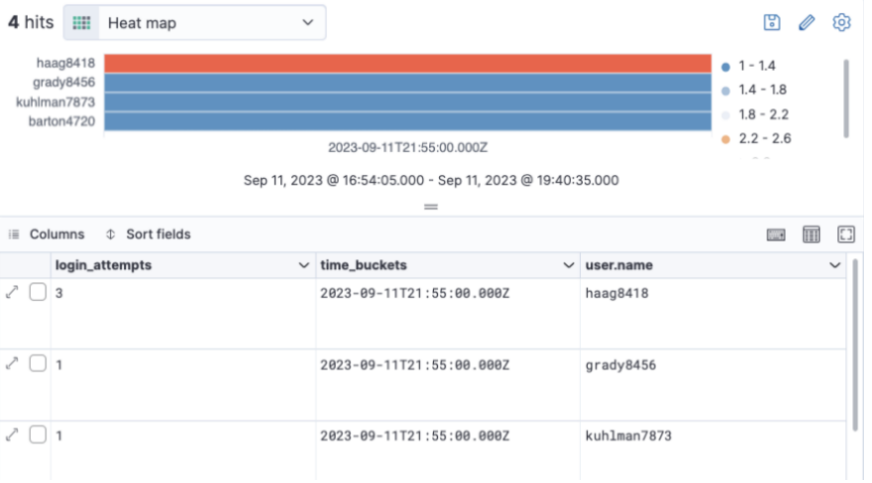
The syntax of ES|QL focuses on a pipe-based structure (|) that guides the flow of data through various operations, facilitating a logical and sequential construction of queries. This methodology allows users to perform complex transformations and data analysis in an intuitive and efficient manner. For example, a query might start with data extraction using FROM, followed by a series of filters and transformations such as WHERE, GROUP BY, and ORDER BY, each connected by a pipe for smooth and orderly execution.

Example:

```
FROM apache-logs
| WHERE url.original == ‘/login’
| EVAL time_buckets = auto_bucket
(@timestamp, 50,”
2023-09-11T21:54:05.000Z”,”
2023-09-12T00:40:35.000Z”)
| STATS login_attempts =
count(user.name) by time_buckets,
user.name
| SORT login_attempts desc
```

In this example, we can see how we select the 'apache-logs' index by filtering messages that contain '/login', evaluate data between two temporal points, distributing them into 50 'buckets', counting the identified events and grouping them by user to finally sort them in descending order.

As a result, we would obtain something like this:



Although this is just an example, it illustrates a security use case that adds significant value.

Later on, we will delve deeper into use cases and examples that will allow us to gradually explore this adventure with ES|QL.

Use Cases of ES|QL

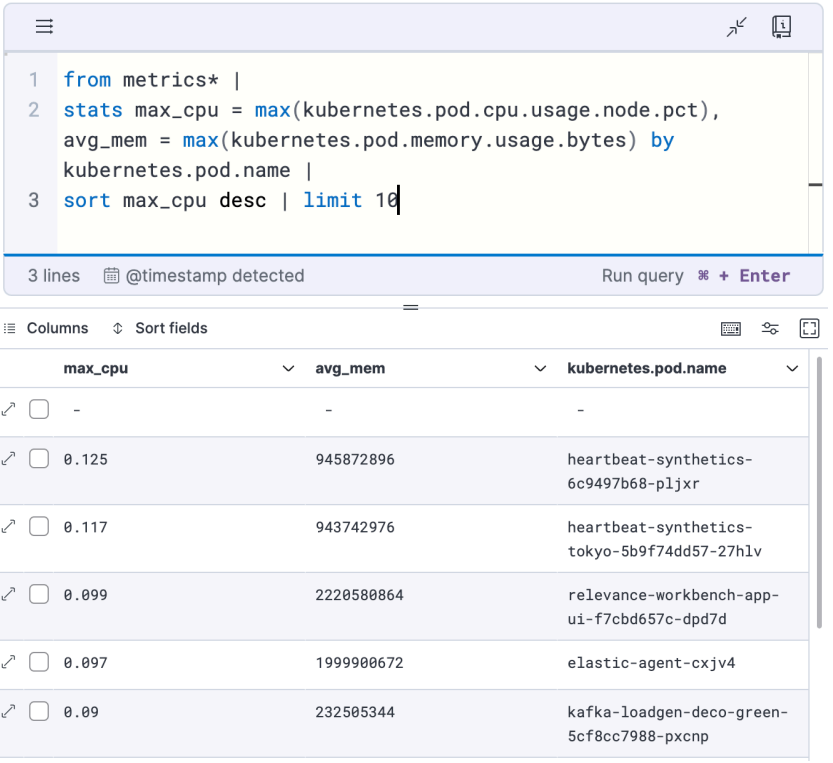
In the previous chapter where we introduced ES|QL and its syntax, we saw a practical example in a security case, where we evaluated the logins of our users.

In this chapter, we will see the three main use cases for ES|QL, although we are sure you can apply them to many others.

Observability

Known as 'o11y' for short, ES|QL will help us by visualizing and analyzing data within Elasticsearch. Directly from the search bar, you can add, transform, calculate, and search your metrics, logs, and traces with a single query to optimize the identification of performance bottlenecks and system issues, reducing resolution time.

ES|QL introduces advanced capabilities such as defining fields at query time, searches for data enrichment, and concurrent query processing, improving speed and efficiency.

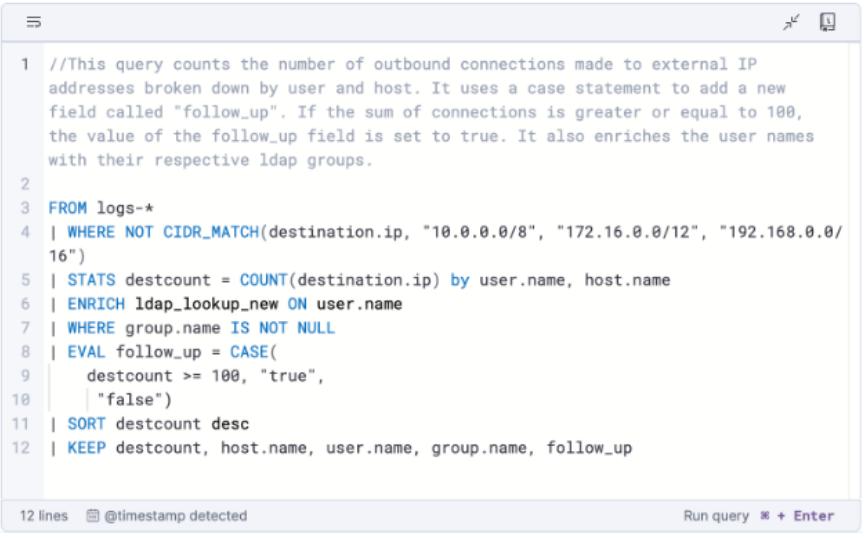


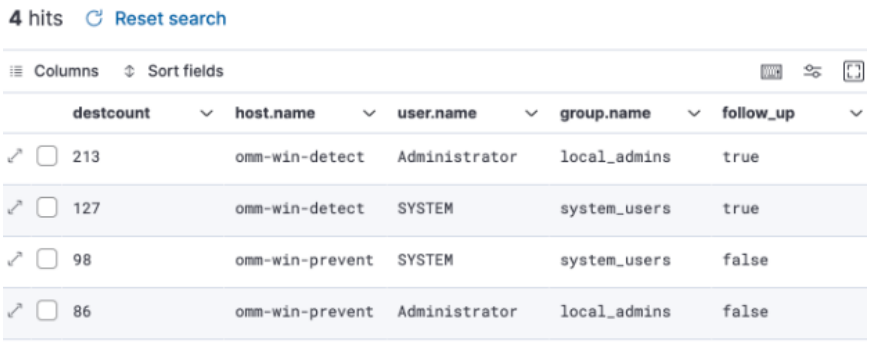
Security

Enhance your security posture, accelerate SecOps workflows, and improve the accuracy of alerts, regardless of the source and structure of the data.

ES|QL allows for quick and flexible searches, defining new fields on the fly, aggregating results, and visualizing significant patterns, all from a single query. It incorporates added values in detection rules to alert more accurately and reduce alarm fatigue, providing more signals and less noise.

Enrich data by providing crucial additional context for security investigations, such as the geographical location of an IP address and its association with malicious entities, the user and their group, all from a single search bar





Search

Elevate your search capabilities, developers will find in ES|QL a simplified coding and querying experience, translating into significant time and cost savings.

ES|QL not only facilitates a greater understanding of the data—what it contains, how to organize it, and how to troubleshoot issues—but also optimizes tasks by consolidating them into a single concurrently processable query, improving performance and reducing the Total Cost of Operation (TCO), allowing more to be achieved with less.

This efficiency is achieved without sacrificing the depth or quality of analysis, offering a comprehensive solution for data manipulation

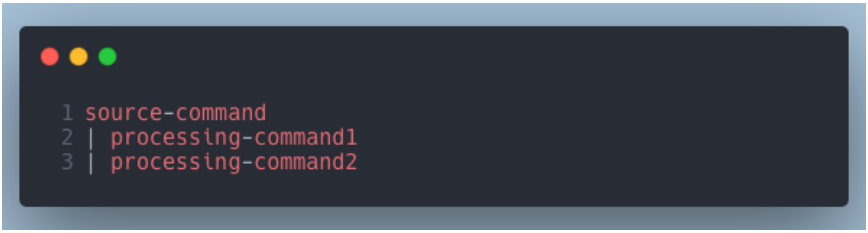
Columns Sort fields

avgbaseprice	category	day_of_week
65	Women's Clothing	Sunday
60	Women's Clothing	Monday
60.833333333333336	Women's Clothing	Tuesday
33	Men's Clothing	Wednesday
61.25	Women's Clothing	Thursday
67.5	Women's Clothing	Friday
65	Women's Clothing	Wednesday

Syntax of ES|QL

In this chapter, we will focus on the syntax of ES|QL and gradually delve deeper into understanding and learning.

In general, ES|QL is composed of a "source-command" followed (optionally) by a series of "processing-commands."



Although everything can be put on a single line, we can stack it to facilitate reading.

Identifiers

They are names used to identify elements such as fields. They can be used as is, except if they contain special characters, in which case they must be enclosed in backticks (`` ` ``).

```
// Retain just one field
FROM index
| KEEP 1.field

// Copy one field
FROM index
| EVAL my_field = `1.field`
```

Literals

Represent fixed values in queries, supporting both numeric and text strings. The latter must be enclosed in double quotes (`""`) and numbers may be integers, decimals, or in scientific notation.

```
// Filter by a string value
FROM index
| WHERE first_name == "Georgi"

ROW name = """Indiana "Indy" Jones"""


1969     -- integer notation
3.14     -- decimal notation
.1234    -- decimal notation starting with decimal point
4E5      -- scientific notation (with exponent marker)
1.2e-3   -- scientific notation with decimal point
-.1e2    -- scientific notation starting with the negative sign
```

Comments

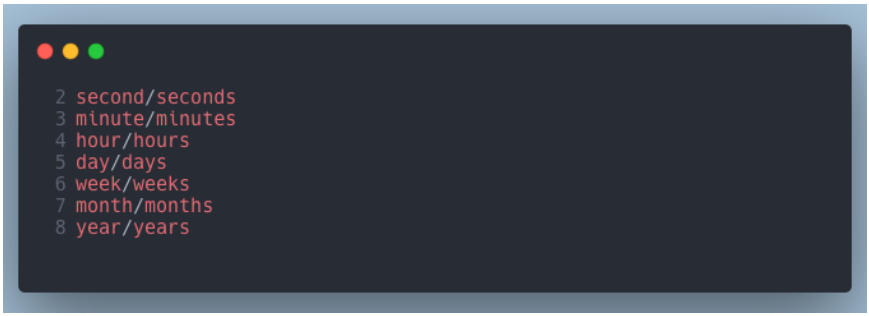
It is allowed to add notes or disable parts of the code, for this we use '//' for single-line comments or '/* ... */' for block comments.

```
// Query the employees index
FROM employees
| WHERE height > 2

FROM /* Query the employees index */ employees
| WHERE height > 2

FROM employees
/* Query the
 * employees
 * index */
| WHERE height > 2
```

Date and time intervals

These can be expressed using interval literals, combining a number with a time qualifier like 'days', 'hours', etc., and can be used to specify durations or periods of time.

List of compatible qualifiers:



```
second/seconds
minute/minutes
hour/hours
day/days
week/weeks
month/months
year/years
```

Commands of ES|QL

There are two categories of commands in ES|QL, known as "source" and "processing" commands..

Source Commands

These commands produce tables and are the start of any query, including:

- FROM
- ROW
- SHOW

Processing Commands

These process and modify the input received by adding, deleting, or changing rows or columns, supported commands include:

- DISSECT
- DROP
- ENRICH
- EVAL
- GROK
- KEEP
- LIMIT
- MV_EXPAND
- RENAME
- SORT
- STATS ... BY
- WHERE

Next, we will delve deeper into each of them. In later chapters, we will see practical examples.

FROM

Specifies the data source for the query, such as an index or alias, and returns a table respecting documents, with rows and columns corresponding to documents and fields.

```
FROM kibana_sample_data_ecommerce
```

ROW

Allows generating a row with one or more columns with specific values. This can be useful for testing.

```
ROW a = 1, b = "two", c = null
```

SHOW

Provides information about the deployment (SHOW INFO) and about functions (SHOW functions) where we can delve deeper as in the example below.

```
SHOW functions
| WHERE STARTS_WITH(name, "sin")
```

DISSECT

Allows the extraction of structured information from a string, where a match of a pattern will be performed as in the example below.

```
ROW a = "2023-01-23T12:15:00.000Z -
some text - 127.0.0.1"
| DISSECT a "%{date} - %{msg} - %{ip}"
```

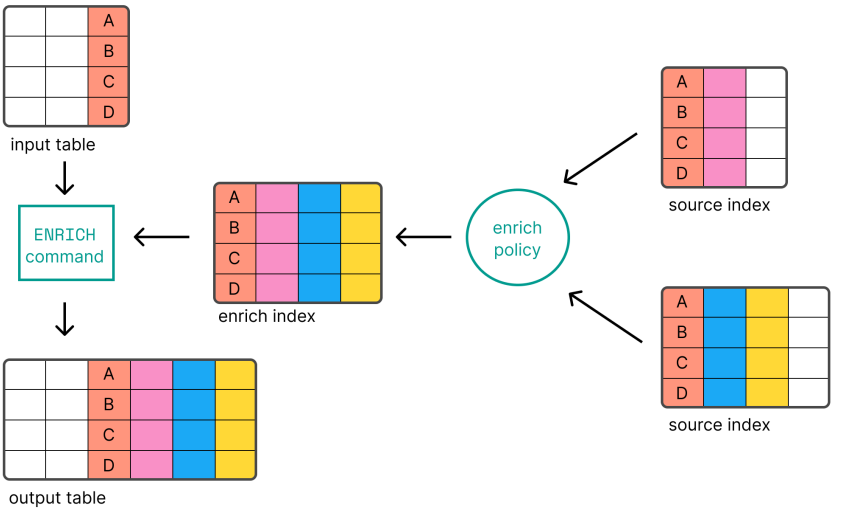
DROP

Can remove columns (fields) that are not needed in the data processing process.

```
ROW a = "2023-01-23T12:15:00.000Z -
some text - 127.0.0.1"
| DISSECT a "%{date} - %{msg} - %{ip}"
| DROP a
```

ENRICH

This command will allow enriching the query process data using enrichment policies, allowing a wide range of possibilities.



EVAL

With this, we can add new columns or fields with calculated values, thanks to the use of Functions

```
from kibana_sample_data_ecommerce
| EVAL price_iva =
mv_sum(products.base_price)* 1.21
| KEEP products.base_price, price_iva
```

GROK

Like DISSECT, it allows the extraction of structured information from a string or text chain, based on a pattern

```
ROW a = "2023-01-23T12:15:00.000Z
127.0.0.1 some.email@foo.com 42"
| GROK a "%{TIMESTAMP_ISO8601:date} %
{IP:ip} %{EMAILADDRESS:email} %
{NUMBER:num:int}”
| KEEP date, ip, email, num
```

KEEP

This command allows us to select from an output, the columns or fields we want to keep, similar to DROP.

```
from kibana_sample_data_ecommerce
| KEEP
customer_full*.keyword,customer_id,
email
```

LIMIT

Limits the number of rows returned by the query (default is 500) allowing finer control over the volume of data processed.

This limit only applies to the number of rows that the query retrieves. The queries and aggregations are executed on the entire dataset.

MV_EXPAND

Expands multi-value columns into a row per value, duplicating other columns.

```
ROW a=[1,2,3], b="b", j=["a","b"]
| MV_EXPAND a
| MV_EXPAND j
```

6 results

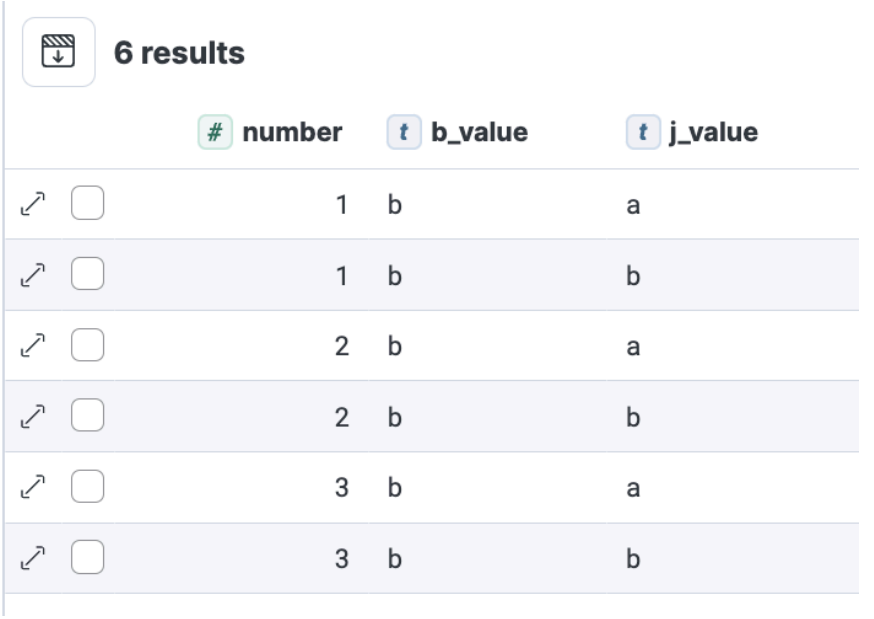
a	b	j
1	b	a
1	b	b
2	b	a
2	b	b
3	b	a
3	b	b

RENAME

Changes the name of one or more columns. If a column already exists with the new name, it will be replaced by the new column.

```
ROW a=[1,2,3], b="b", j=["a","b"]
| MV_EXPAND a
| MV_EXPAND j
```

```
| RENAME a as number, b as b_value, j
as j_value
```



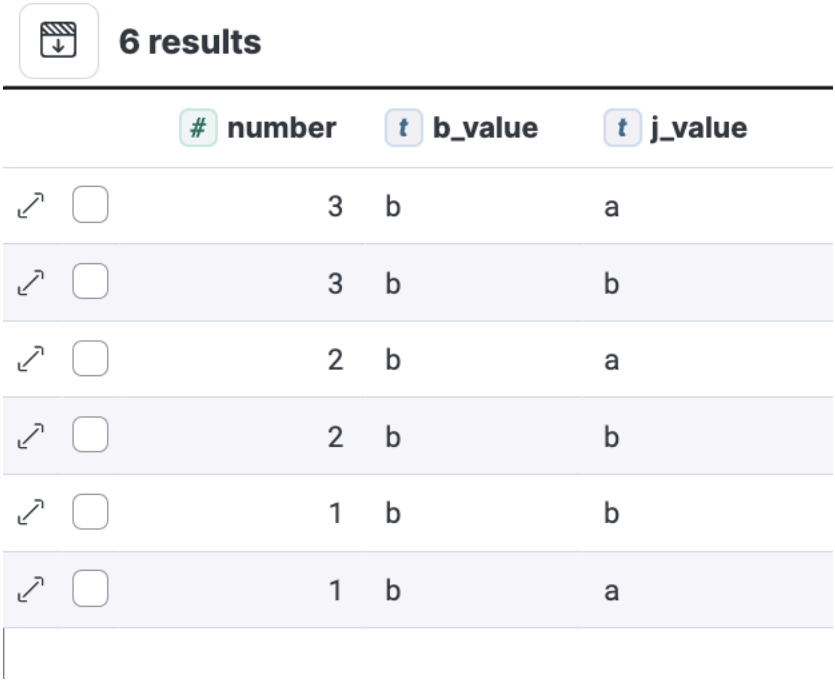
6 results

	# number	t b_value	t j_value
⤢ ☐	1	b	a
⤢ ☐	1	b	b
⤢ ☐	2	b	a
⤢ ☐	2	b	b
⤢ ☐	3	b	a
⤢ ☐	3	b	b

SORT

Allows sorting the table or one or more columns

```
ROW a=[1,2,3], b="b", j=["a","b"]
| MV_EXPAND a
| MV_EXPAND j
| RENAME a as number, b as b_value, j
as j_value
| SORT number desc
```



STATS ... BY

Groups rows according to a common value and calculates one or more aggregated values over the grouped rows. If BY is omitted, the output table contains exactly one row with the aggregations applied to the entire dataset.

```
FROM kibana_sample_data_ecommerce
| EVAL price_iva =
mv_sum(products.base_price)* 1.21
| KEEP products.base_price, price_iva,
customer_id, customer_first_name
| STATS avg(price_iva) BY
customer_first_name
```

46 results

avg(price_iva)	customer_first_name
84.45894531249999	Abdulraheem Al
116.399211328125	Eddie
84.80904854910713	Mary
84.74110559682377	Gwen

WHERE

Produces a table that contains all rows from the input table for which the provided condition evaluates to true.

```
FROM kibana_sample_data_ecommerce
| EVAL price_iva =
mv_sum(products.base_price)* 1.21
| KEEP products.base_price, price_iva,
customer_id, customer_first_name
| STATS avg(price_iva) BY
customer_first_name
| WHERE `avg(price_iva)` < 100 and
customer_first_name RLIKE "M.*"
```

4 results

avg(price_iva)	customer_first_name
84.80904854910713	Mary
93.54259732521186	Muniz
83.45342051630435	Mostafa
80.06774719238281	Marwan

Functions and Operators of ES|QL

The ES|QL functions and operators allow you to manipulate and analyze data efficiently and flexibly, facilitating the creation of queries to search for specific events, perform statistical analyses, and generate visualizations.

Functions

Mainly we have:

Aggregation: AVG, COUNT, COUNT_DISTINCT, MAX, MEDIAN, MEDIAN_ABSOLUTE_DEVIATION, MIN, PERCENTILE, ST_CENTROID, SUM.

Mathematical: ABS, ACOS, ASIN, ATAN, ATAN2, CEIL, COS, COSH, E, FLOOR, LOG, LOG10, PI, POW, ROUND, SIN, SINH, SQRT, TAN, TANH, TAU.

Text Strings: CONCAT, LEFT, LENGTH, LTRIM, REPLACE, RIGHT, RTRIM, SPLIT, SUBSTRING, TO_LOWER, TO_UPPER, TRIM.

Dates and Times: AUTO_BUCKET, DATE_DIFF, DATE_EXTRACT, DATE_FORMAT, DATE_PARSE, DATE_TRUNC, NOW.

Type Conversion: TO_BOOLEAN, TO_CARTESIANPOINT, TO_CARTESIANSHAPE, TO_DATETIME, TO_DEGREES, TO_DOUBLE, TO_GEOPOINT, TO_GEOSHAPE, TO_INTEGER, TO_IP, TO_LONG, TO_RADIANS, TO_STRING, TO_UNSIGNED_LONG, TO_VERSION.

Conditional Functions and Expressions: CASE, COALESCE, GREATEST, LEAST.

Multi-value Functions: MV_AVG, MV_CONCAT, MV_COUNT, MV_DEDUPE, MV_FIRST, MV_LAST, MV_MAX, MV_MEDIAN, MV_MIN, MV_SUM.

Operators

Binary (==, !=, <, <=, >, >=, +, -, *, /, %), Unary (-), Logical (AND, OR, NOT), IS NULL / IS NOT NULL, CIDR_MATCH, ENDS_WITH, IN, LIKE, RLIKE, STARTS_WITH.

Metadata fields of ES|QL

We can access metadata fields (_index, _id, _version) from the FROM command, which is the only directive that supports this field.

In possible use cases, accessing _id allows us to perform operations when we already know the value of this field, or if it is common among indices and we want to perform joint operations. Similarly, with _index, for example, when we need to perform groupings by indices.

```
from kibana* metadata _id, _index
| stats n_documents = count(_id) by
_index
```

3 results

n_documents	_index
14074	.ds-kibana_sample_data_logs-2024.01.29-000001
13014	kibana_sample_data_flights
4675	kibana_sample_data_ecommerce

Multi-valued fields in ES|QL

With ES|QL, we can also operate on fields with multiple values, though we need to consider the data type. Keywords remove duplicates, so if we index a document with field X containing values ["a", "b", "a"], the result will be ["a", "b"]. However, if we use a field of type long, for example, "x": ["1", "2", "1"], duplicates will be retained.

Processing Data with GROK and DISSECT

When the data we work with contains unstructured information, we can use GROK and DISSECT to parse it, for example:

Elasticsearch can structure data at indexing time or at query time, and in both cases, we can use the well-known GROK and DISSECT.

Choosing which one to use will depend on our data and the type of analysis we want to perform. Sometimes DISSECT will be much more efficient if our data is delimited, while we will use GROK if we want to use regular expressions.

GROK is the most powerful but also the slowest
DISSECT works well when the data is repetitive (without exceptions)

```
ROW a = "2023-01-23T12:15:00.000Z - some text - 127.0.0.1"
| DISSECT a "%{date} - %{msg} - %{ip}"
| KEEP date, msg, ip
```

date	msg	ip
2023-01-23T12:15:00.000Z	some text	127.0.0.1

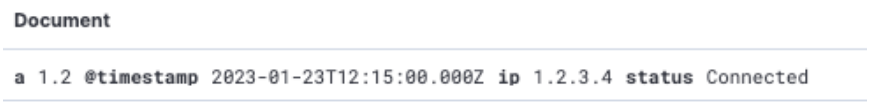
Another example where we handle the date:

```
ROW a = "2023-01-23T12:15:00.000Z - some text - 127.0.0.1"
| DISSECT a "%{date} - %{msg} - %{ip}"
| KEEP date, msg, ip
| EVAL date = TO_DATETIME(date)
```

msg	ip	date
some text	127.0.0.1	2023-01-23T12:15:00.000Z

And on the other side, we have GROK:

```
ROW a = "1.2.3.4
[2023-01-23T12:15:00.000Z] Connected"
| GROK a "%{IP:ip} \\[%
{TIMESTAMP_ISO8601:@timestamp}\\] %
{GREEDYDATA:status}"
```



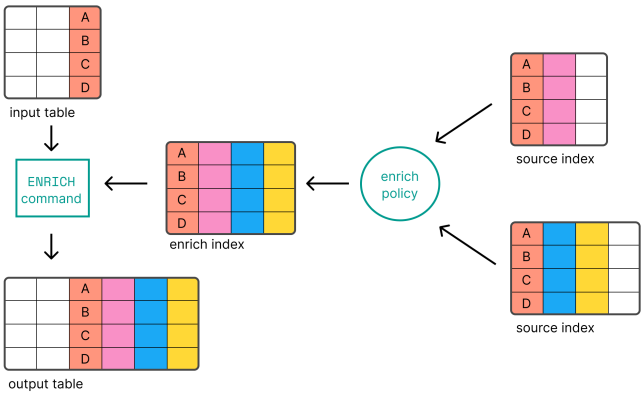
We have some limitations when using GROK with ES|QL. "Custom patterns" or "multiple patterns" cannot be used, and it is not subject to the "GROK Watchdog," which interrupts the execution of GROK if it exceeds a certain time.

Even though we can process information at query time, it is always good, once the scope and patterns have been identified, to process or preprocess the data at the time of ingestion.

Enriching Data

As we have seen earlier, we have a processing command that allows us to enrich data at query time, "ENRICH," and now we will look at it in more detail.

Data enrichment is a process by which, based on previously identified fields, we combine data from different indices:



Enrichment Policy

In the policy, we define one or more source indices from which we will take the data. It is determined how the data processed by this policy will be matched and which fields from the source index will be added in the enrichment

Source Index

This is a conventional or regular index, called the source because it contains the data that we will add to the information processing.

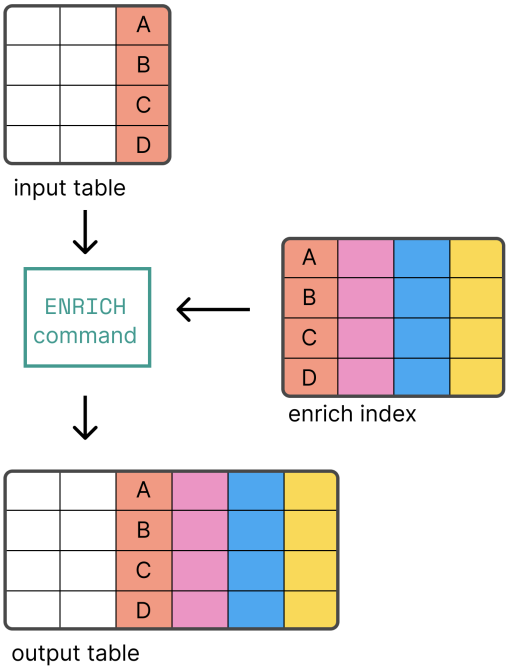
Enrichment Index

his is a special system index generated when we execute the enrichment policy. It always starts with “.enrich-*”, is read-only, and cannot be modified directly. A "force merge" is performed on it to improve query times.

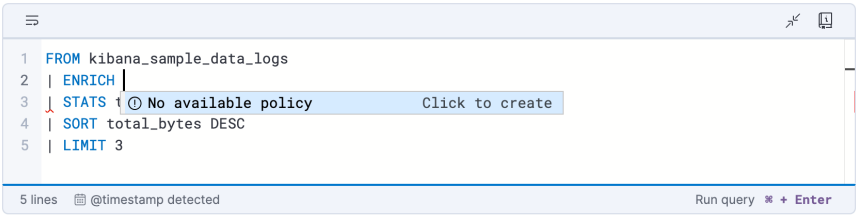
If we make any changes or updates to the source index and want them to be reflected in the enrichment, we must re-execute the enrichment policy so that these changes are reflected in the enrichment index.

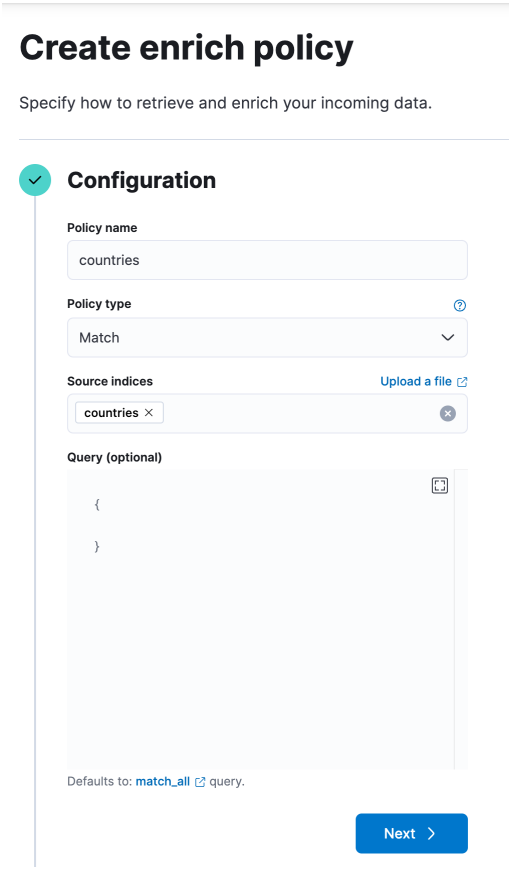
Using the Enrich Policy

After defining and executing the enrichment policy, we can use it:



Y también la podemos definir y crear desde Kibana cuando necesitemos usarla:


FROM kibana_sample_data_logs
| ENRICH
| STATS t
No available policy
Click to create
| SORT total_bytes DESC
| LIMIT 3
5 lines
@timestamp detected
Run query ⌘ + Enter


Create enrich policy
Specify how to retrieve and enrich your incoming data.
Configuration
Policy name
countries
Policy type
Match
Source indices
Upload a file
countries
Query (optional)
{
}
Defaults to: match_all query.
Next

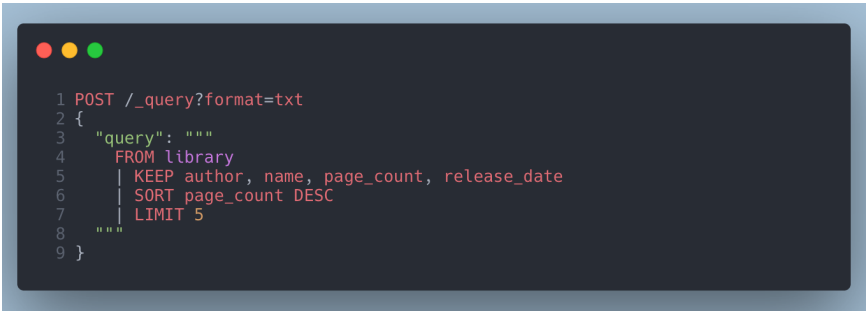
Using ES|QL

We have different ways to use ES|QL, including the REST API, Kibana's Discover, Kibana's Security section, and across multiple clusters.

ES|QL query API

We can use it from the REST API as shown in the following example:

Also from Kibana's "Dev Tools":



```
POST /_query?format=txt
{
  "query": """
    FROM library
    | KEEP author, name, page_count, release_date
    | SORT page_count DESC
    | LIMIT 5
  """
}
```

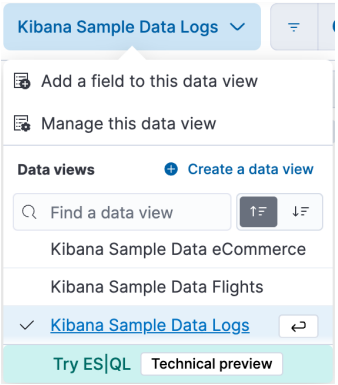
IWe can even apply custom "Query DSL" filters, thus limiting the scope or set of documents where ES|QL will be executed:

```
POST /_query?format=txt
{
  "query": """
    FROM library
    | KEEP author, name, page_count, release_date
    | SORT page_count DESC
    | LIMIT 5
  """,
  "filter": {
    "range": {
      "page_count": {
        "gte": 100,
        "lte": 200
      }
    }
  }
}

```

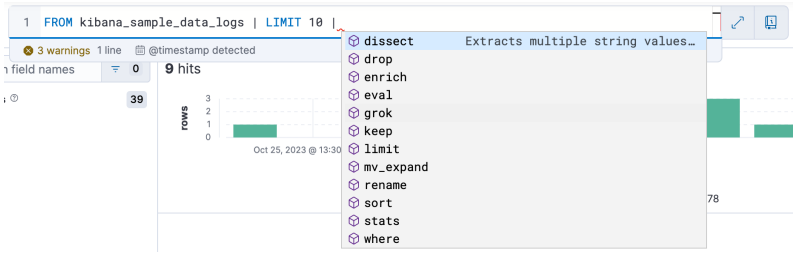
We can also request it to return the data in columns, define the "locale," pass parameters, and execute asynchronous queries.

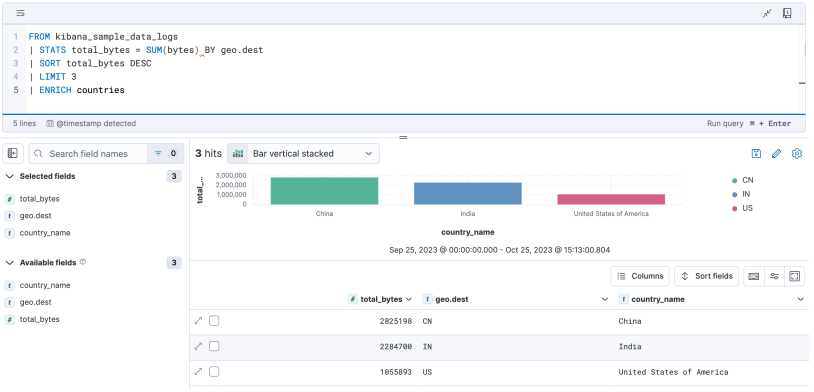
Using ES|QL in Kibana

From Discover, we can indicate in the "Data view" section that we want to use ES|QL:

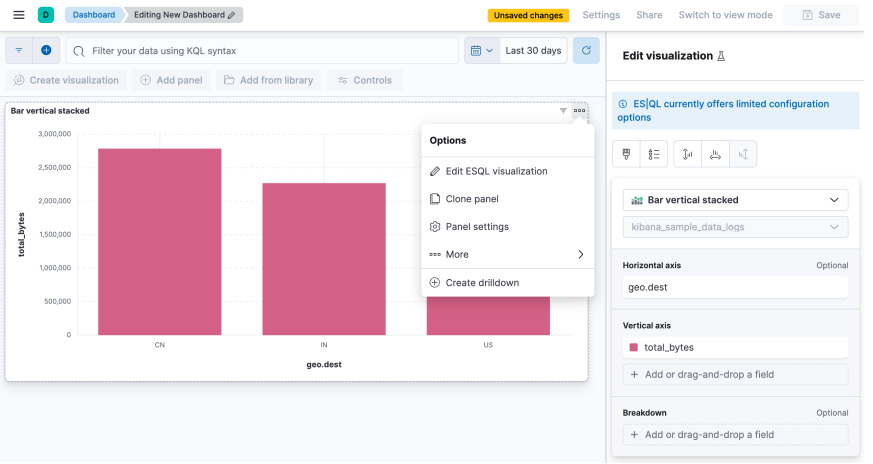


This way, we can use the query bar to run our ES|QL queries





We can also generate graphs/visualizations to use later in “dashboards”



Managing ES|QL

There may be times when we run queries that are consuming too many resources, are too slow, or take too long to return results, and we need to cancel requests that are in flight.

For this, we will use the task manager as follows:

```
GET /_tasks?pretty&detailed&group_by=parents&human&actions=*data/read/esql
```

This will return something like:

```
{
  "node" : "2j8UKw1bRO283PMwDugNNg",
  "id" : 5326,
  "type" : "transport",
  "action" : "indices:data/read/esql",
  "description" : "FROM test | STATS MAX(d) by a, b",
  "start_time" : "2023-07-31T15:46:32.328Z",
  "start_time_in_millis" : 1690818392328,
  "running_time" : "41.7ms",
  "running_time_in_nanos" : 41770830,
  "cancellable" : true,
  "cancelled" : false,
  "headers" : { }
}
```

And we can act on that task like any other in the following way:

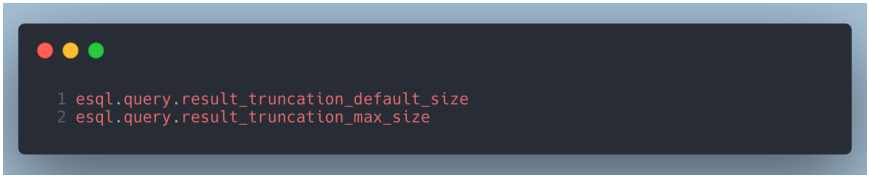
```
POST _tasks/2j8UKw1bRO283PMwDugNNg:5326/_cancel
```

Limitations

There are several limitations, many of them inherited, such as the limitation on the number of returned data, which is 1,000 by default and up to 10,000 at maximum. These limitations do not apply when performing aggregations that are executed over the entire data set.

To use this correctly and avoid reaching this limitation regarding the data limit, we will focus on using “WHERE” and “STATS ... BY”

Although we can dynamically change the default value with the following parameters, it is always advisable to study the use case to avoid performance loss.



Supported Field Types

At the time of writing this handbook, the supported field types are:

- alias
- boolean
- date
- double (float, half_float, scale_float representados como double)
- ip
- keyword
- int (sort, byte representados como int)
- long
- null
- text
- unsigned_long
- version
- geo_point
- geo_shape
- point
- shape

Unsupported Field Types

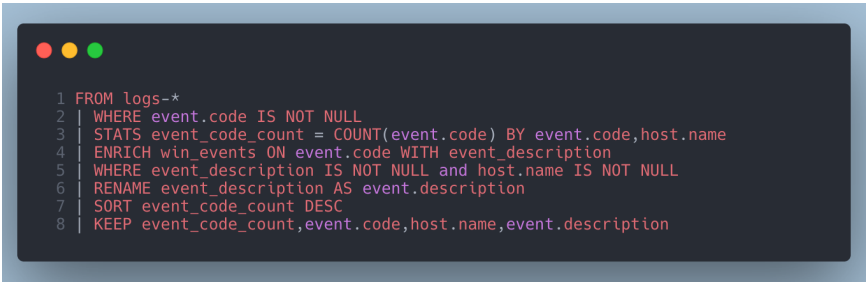
At the time of writing this handbook, the unsupported field types are:

- TBS metrics:
  - counter
  - position
  - aggregare_metric_double
- Date/time:
  - date_nanos
  - date_range
- Other types:
  - binary
  - completion
  - dense_vector
  - double_range
  - flattened
  - float_range
  - histogram
  - integer_range
  - ip_range
  - long_range
  - nested
  - rank_feature
  - rank_features
  - search_as_you_type

Examples

Next, we will see several examples, and in later chapters, we will have resources for day-to-day use of ES|QL

Aggregating and enriching Windows event logs:



Summing the outgoing traffic of a specific process (curl.exe)

```
FROM logs-endpoint
| WHERE process.name == "curl.exe"
| STATS bytes = SUM(destination.bytes) BY destination.address
| EVAL kb =  bytes/1024
| SORT kb DESC
| LIMIT 10
| KEEP kb,destination.address
```

Processing DNS logs to find a high number of unique DNS queries per registered domain.

```
FROM logs-*
| GROK dns.question.name "%{DATA}\\.%
{GREEDYDATA:dns.question.registered_domain:string}"
| STATS unique_queries = COUNT_DISTINCT(dns.question.name) BY
dns.question.registered_domain, process.name
| WHERE unique_queries > 10
| SORT unique_queries DESC
| RENAME unique_queries AS `Unique Queries`, dns.question.registered_domain
AS `Registered Domain`, process.name AS `Process`
```

Identifying a high number of outgoing connections from users.

```
FROM logs-*
| WHERE NOT CIDR_MATCH(destination.ip, "10.0.0.0/8", "172.16.0.0/12",
"192.168.0.0/16")
| STATS destcount = COUNT(destination.ip) BY user.name, host.name
| ENRICH ldap_lookup_new ON user.name
| WHERE group.name IS NOT NULL
| EVAL follow_up = CASE(destcount >= 100, "true","false")
| SORT destcount DESC
| KEEP destcount, host.name, user.name, group.name, follow_up
```

Resources

In this section, we will look at several resources that will help you in using ES|QL

GitHub for practical examples:

https://github.com/elastic/elasticsearch/tree/main/x-pack/plugin/esql/qa/testFixtures/src/main/resources

ES|QL demo environment:

https://esql.demo.elastic.co

ES|QL Quick Reference