# **Tutorial: Search and Analyze data in Amazon OpenSearch using SQL**

OpenSearch is a distributed search and analytics technology in the database family, providing a REST api to send data and query that data. It helps everyone find what they need faster—from employees who need documents from their intranet to customers browsing online for the perfect pair of shoes.

The speed and scalability of OpenSearch and its ability to index many types of content mean that it can be used for a number of use cases including Website search, Enterprise search, Log analytics, Application performance monitoring and Security analytics.

Unlike other database technologies, you structure your data in indexes instead of tables. Each json object that you send to you OpenSearch index is called a 'document' (c.f. 'row' in a relational database). The JSON keys for that document are called 'fields' (c.f. columns), and the values are values.

Amazon OpenSearch is a community-driven, open source fork of Elasticsearch and Kibana. AWS had made a long-term investment in OpenSearch to ensure users continue to have a secure, high-quality, fully open source search and analytics suite with a rich roadmap of new and innovative functionality. This project includes OpenSearch (derived from Elasticsearch 7.10.2) and OpenSearch Dashboards (derived from Kibana 7.10.2). All of the software in the OpenSearch project is released under the Apache License, Version 2.0 (ALv2).

Amazon OpenSearch Service offers a choice of open source engines to deploy and run, including multiple versions of ALv2 Elasticsearch as well as new versions of OpenSearch. AWS continues to support and maintain the ALv2 Elasticsearch versions with security and bug fixes, and but all new features and functionality are delivered through OpenSearch and OpenSearch Dashboards. The Amazon OpenSearch Service APIs will be backward compatible with the existing service APIs to eliminate any need for customers to update their current client code or applications. Additionally, AWS provides a seamless upgrade path from existing Elasticsearch 6.x and 7.x managed clusters to OpenSearch.

In this tutorial you'll get hands on with using SQL with Amazon OpenSearch Serverless. You will walk through setting up a new Amazon OpenSearch Serverless domain in the AWS console. You'll explore the different types of search queries available. You'll design eye-catching visualizations, and learn how you can secure your domain and documents based on assigned user privileges. The target audience for this tutorial includes developers and members of Information Technology departments. You also learn how to create and search for a document in Amazon OpenSearch Service. When you add data to an index in the form of a JSON document the OpenSearch Service creates an index around the first document that you add.

###### Note

This tutorial uses a domain with open access. For the highest level of security, we recommend that you put your domain inside a virtual private cloud (VPC).

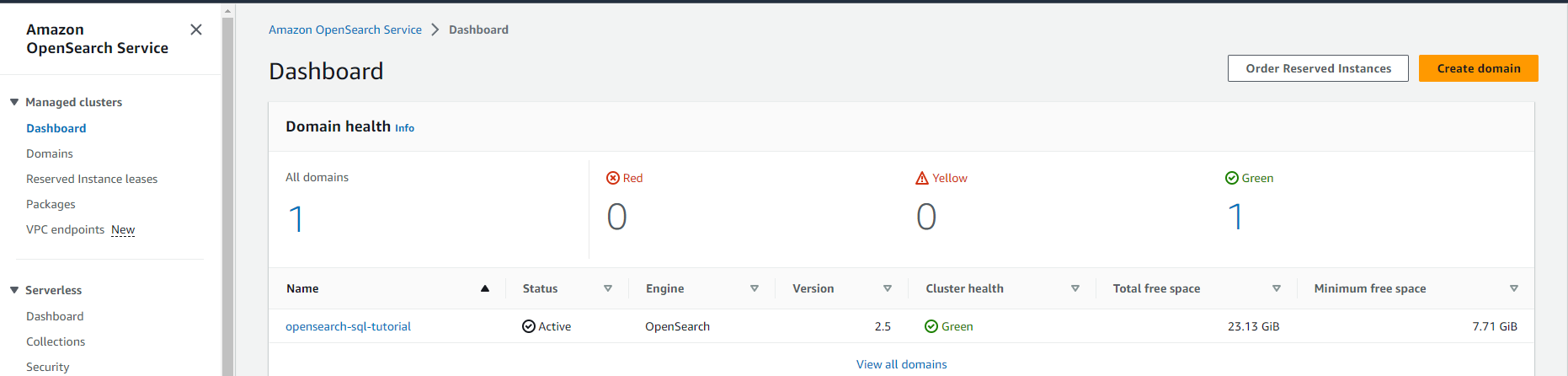
## Prerequisites

This tutorial has the following prerequisites:

* You must have an AWS account.
* You must have an active OpenSearch Service domain.

**Step 1. Ingest Sample data into your OpenSearch domain.**

1a. Navigate to your OpenSearch service on the AWS console. On the Dashboard section, your domain should be listed under the header “Name”. Click on any of your domains.



1b. Click on the OpenSearch Dashboard URL link for this domain.

## 

1c. Click on the “Add Sample data” link.

## 

1d. Click on the “Add data” button for Sample eCommerce Orders (if it is already installed you will see “View data” instead, you can skip this step in that case).

## 

1d. Click the Hamburger Icon to expand the menu, and then click on “Query Workbench”. This will open up the “Query Editor” where you can enter your SQL queries, with the results showing in the “Output” pane below that.

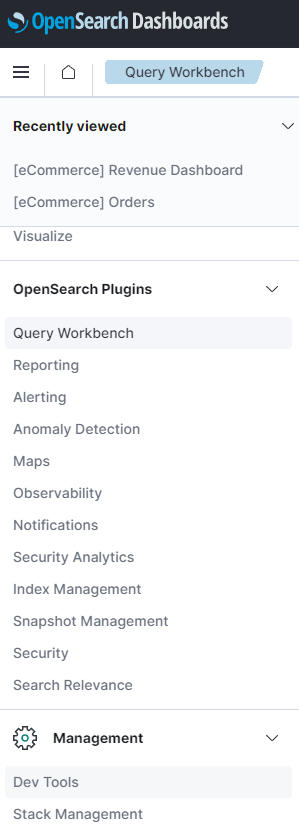
## 

## 

**Step 2. (Optional) Add documents to an index into your OpenSearch domain.**

To add a document to an index, you can use any HTTP tool, such as [Postman](https://www.getpostman.com/), cURL, or the OpenSearch Dashboards console. These examples assume that you’re using the developer console in OpenSearch Dashboards. If you’re using a different tool, adjust accordingly by providing the full URL and credentials, if necessary.

2a. Navigate to the OpenSearch Dashboards URL for your domain (same as above). Open the left navigation panel and choose **Dev Tools**.



1. The HTTP verb for creating a new resource is PUT, which is what you use to create a new document with an ID of “1” and creates a new index. Enter the following command in the console:

PUT fruit/\_doc/1

{

"name":"strawberry",

"color":"red"

}

The PUT request creates an index named fruit and adds a single document to the index with an document ID of 1. It produces the following response:

{

"\_index" : "fruit",

"\_type" : "\_doc",

"\_id" : "1",

"\_version" : 1,

"result" : "created",

"\_shards" : {

"total" : 2,

"successful" : 2,

"failed" : 0

},

"\_seq\_no" : 0,

"\_primary\_term" : 1

}

OpenSearch can automatically generate an ID for your documents if you use a POST request instead of a PUT request, and it requires no document ID (in comparison to the previous request).

Enter the following request in the developer console:

POST fruit/\_doc

{

"name":"peanut",

"color":"brown",

"classification":"underground"

}

**Step 3: Mapping concepts across SQL and OpenSearch**

While SQL and OpenSearch have different terms for the way the data is organized (and different semantics), essentially their purpose is the same.So let’s start from the bottom; these roughly are:

| **SQL** | **OpenSearch** | **Description** |
| --- | --- | --- |
| column | field | In both cases, at the lowest level, data is stored in named entries, of a variety of data types, containing one value. SQL calls such an entry a column while OpenSearch a field. Notice that in OpenSearch a field can contain multiple values of the same type (essentially a list) while in SQL, a column can contain exactly one value of said type. OpenSearch SQL will do its best to preserve the SQL semantic and, depending on the query, reject those that return fields with more than one value. |
| row | document | Columns and fields do not exist by themselves; they are part of a row or a document. The two have slightly different semantics: a row tends to be strict (and have more enforcements) while a document tends to be a bit more flexible or loose (while still having a structure). |
| table | index | The target against which queries, whether in SQL or OpenSearch get executed against. |
| schema | implicit | In RDBMS, schema is mainly a namespace of tables and typically used as a security boundary. OpenSearch does not provide an equivalent concept for it. However when security is enabled, OpenSearch automatically applies the security enforcement so that a role sees only the data it is allowed to (in SQL jargon, its schema). |
| catalog or database | cluster instance or domain | In SQL, catalog or database are used interchangeably and represent a set of schemas that is, a number of tables. In OpenSearch the set of indices available are grouped in a cluster or domain. The semantics also differ a bit; a database is essentially yet another namespace (which can have some implications on the way data is stored) while an OpenSearch cluster is a runtime instance, or rather a set of at least one OpenSearch instance (typically running distributed). In practice this means that while in SQL one can potentially have multiple catalogs inside an instance, in OpenSearch one is restricted to only one. |
| cluster | cluster (federated) | Traditionally in SQL, cluster refers to a single RDMBS instance which contains a number of catalogs or databases (see above). While RDBMS tend to have only one running instance, on a single machine (not distributed), OpenSearch goes the opposite way and by default, is distributed and multi-instance. Further more, an OpenSearch cluster can be connected to other clusters in a federated fashion thus cluster means:  single cluster:: Multiple Elasticsearch instances typically distributed across machines, running within the same namespace. multiple clusters:: Multiple clusters, each with its own namespace, connected to each other in a federated setup.  Cross-cluster search in Amazon OpenSearch Service lets you perform queries and aggregations across multiple connected domains. It often makes more sense to use multiple smaller domains instead of a single large domain, especially when you're running different types of workloads. |

Step 4: Running Basic SQL queries

List all your indexes

SHOW TABLES LIKE %

Results

| **TABLE\_NAME** |
| --- |
|  | .kibana\_1 |
|  | .opendistro\_security |
|  | fruit |
|  | opensearch\_dashboards\_sample\_data\_ecommerce |
|  | opensearch\_dashboards\_sample\_data\_flights |
|  | opensearch\_dashboards\_sample\_data\_logs |
|  | .kibana |

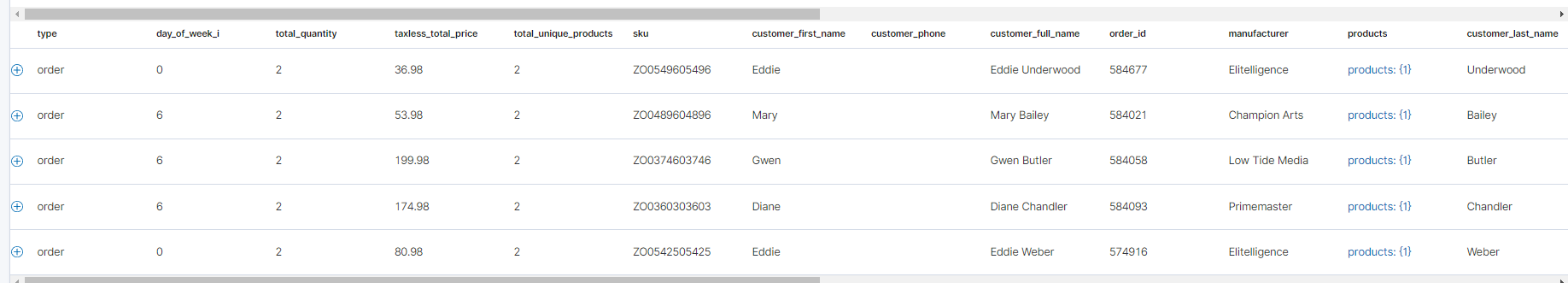
Retrieve a limited set of documents (5)

SELECT \*

FROM opensearch\_dashboards\_sample\_data\_ecommerce

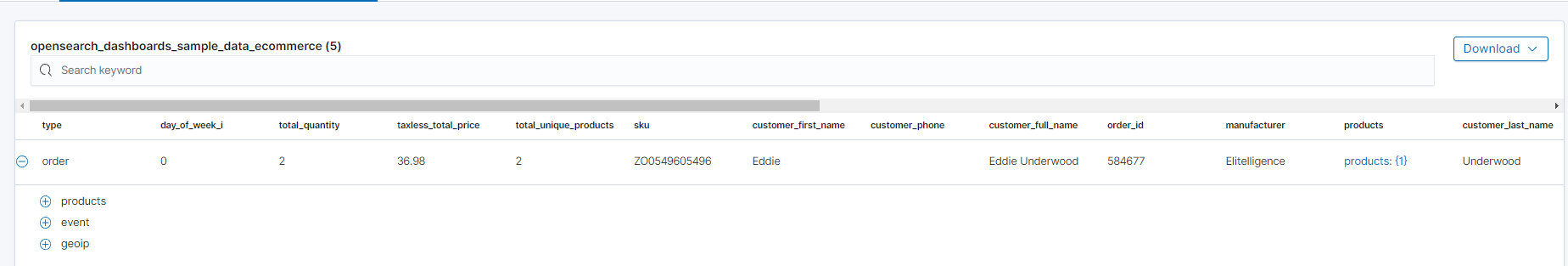
limit 5

Results

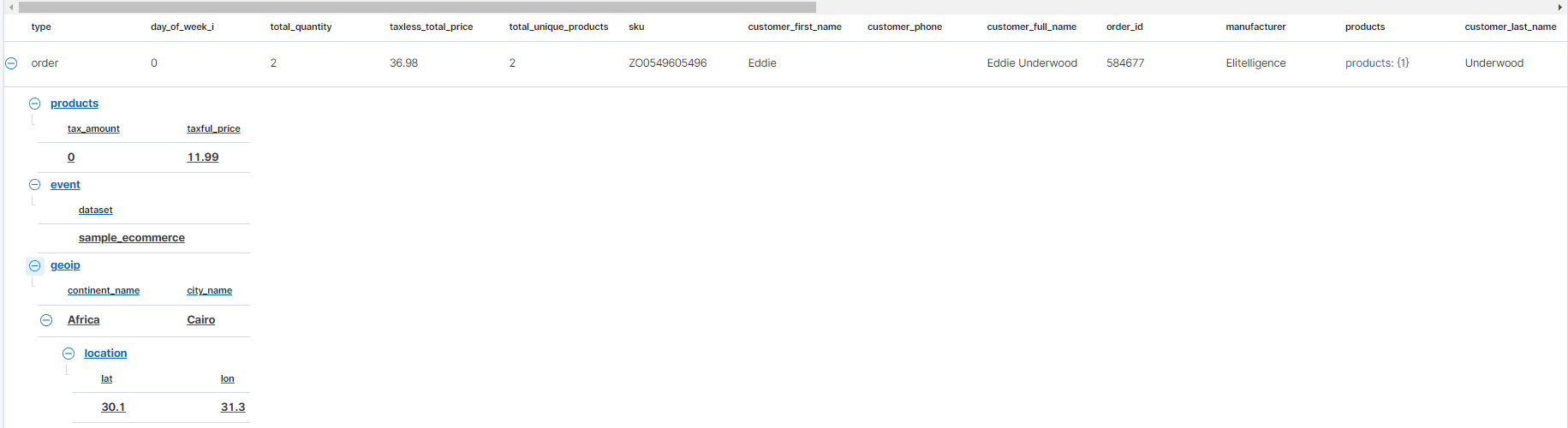


Note – the (+) sign next to “order” indicates that there are nested JSON documents which we can see when we expand it. Later we will see how we can query nested JSON document fields using SQL.

Expanded Results (level 1)



Expanded Results (all levels, for order id 584677)



We want retrieve a specific order, so we need to get the datatypes of all the columns in this table, using the SQL query below, and scrolling through the results to find the datatype of the column order\_id.

DESCRIBE TABLES LIKE opensearch\_dashboards\_sample\_data\_ecommerce



Since the datatype of the column order\_id is keyword, we need to enclose the search in quotes.

SELECT \*

FROM opensearch\_dashboards\_sample\_data\_ecommerce

where order\_id = '584677'

Use the SQLSELECT clause, along with FROM, WHERE, GROUP BY, HAVING, ORDER BY, and LIMIT to search and aggregate data.

Among these clauses, SELECT and FROM are required, as they specify which fields to retrieve and which indexes to retrieve them from. All other clauses are optional.

The complete syntax for searching and aggregating data is as follows:

SELECT [DISTINCT] (\* | expression) [[AS] alias] [, ...]

FROM index\_name

[WHERE predicates]

[GROUP BY expression [, ...]

[HAVING predicates]]

[ORDER BY expression [IS [NOT] NULL] [ASC | DESC] [, ...]]

[LIMIT [offset, ] size]

Use the DISTINCT clause to get back only unique field values. You can specify one or more field names:

SELECT DISTINCT manufacturer

FROM opensearch\_dashboards\_sample\_data\_ecommerce

manufacturer |

-------------------------+

|

Angeldale |

Champion Arts |

Crystal Lighting |

Elitelligence |

Gnomehouse |

Gnomehouse mom |

Use the GROUP BY clause to define subsets of a result set. You can specify the field name (column name) to aggregate on in the GROUP BY clause. For example, the following query returns the department numbers and the total sales for each department:

SELECT department, sum(sales)

FROM employees

GROUP BY department;

|  | **department** | **sum(sales)** |
| --- | --- | --- |
|  | 1 | 58698 |
|  | 2 | 37018 |

You can use an expression in the GROUP BY clause. For example, the following query returns the average sales for each year:

SELECT year(sale\_date), avg(sales)

FROM employees

GROUP BY year(sale\_date);

| **year(start\_date)** | **avg(sales)** |
| --- | --- |
| 2020 | 1356.0 |
| 2021 | 22455.0 |
| 2022 | 16484.0 |

Use aggregate expressions as part of larger expressions in SELECT. The following query calculates the average commission for the employees of each department as 5% of the average sales:

SELECT department, avg(sales) \* 0.05 as avg\_commission

FROM employees

GROUP BY department;

| **department** | **avg\_commission** |
| --- | --- |
| 1 | 733.75 |
| 2 | 925.45 |

Using WHERE and HAVING clauses in SQL

Both WHERE and HAVING are used to filter results. The WHERE filter is applied before the GROUP BY phase, so you cannot use aggregate functions in a WHERE clause. However, you can use the WHERE clause to limit the rows to which the aggregate is then applied. The HAVING filter is applied after the GROUP BY phase, so you can use the HAVING clause to limit the groups that are included in the results.

The following query returns the number of sales for each employee who made more than one sale:

SELECT employee\_id, count(sales)

FROM employees

GROUP BY employee\_id

HAVING count(sales) > 1;

| **employee\_id** | **count(sales)** |
| --- | --- |
| 1 | 2 |
| 6 | 2 |

Use an alias for an aggregate expression in the HAVING clause. The following query returns the total sales for each department where sales exceed $40,000:

SELECT department, sum(sales) as total

FROM employees

GROUP BY department

HAVING total > 40000;

| **department** | **total** |
| --- | --- |
| 1 | 58698 |

The DELETE statement deletes documents that satisfy the predicates in the WHERE clause. If you don’t specify the WHERE clause, all documents are deleted. It is disabled by default. To enable the DELETE functionality in SQL, you need to update the configuration by sending the following request:

PUT \_plugins/\_query/settings

{

"transient": {

"plugins.sql.delete.enabled": "true"

}

}

Then run the following query

DELETE FROM accounts

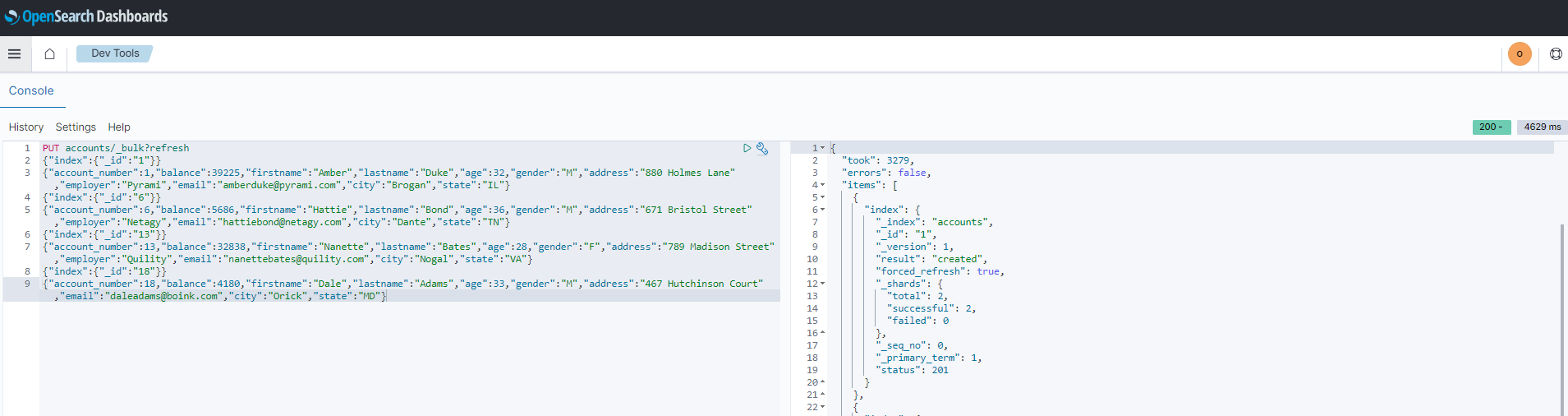
WHERE age > 30

The deleted\_rowsfield shows the number of documents deleted.

**Step 5: Running Complex SQL queries**

Let’s insert some data related to accounts and employees (into their respective indexes) to illustrate some of the SQL join capabilities.

Navigate to DevTools and bulk insert via the following statements. Enter these on the left pane of the console and the results will be shown on the right side of the console.



Statement for inserting into accounts:

PUT accounts/\_bulk?refresh

{"index":{"\_id":"1"}}

{"account\_number":1,"balance":39225,"firstname":"Amber","lastname":"Duke","age":32,"gender":"M","address":"880 Holmes Lane","employer":"Pyrami","email":"amberduke@pyrami.com","city":"Brogan","state":"IL"}

{"index":{"\_id":"6"}}

{"account\_number":6,"balance":5686,"firstname":"Hattie","lastname":"Bond","age":36,"gender":"M","address":"671 Bristol Street","employer":"Netagy","email":"hattiebond@netagy.com","city":"Dante","state":"TN"}

{"index":{"\_id":"13"}}

{"account\_number":13,"balance":32838,"firstname":"Nanette","lastname":"Bates","age":28,"gender":"F","address":"789 Madison Street","employer":"Quility","email":"nanettebates@quility.com","city":"Nogal","state":"VA"}

{"index":{"\_id":"18"}}

{"account\_number":18,"balance":4180,"firstname":"Dale","lastname":"Adams","age":33,"gender":"M","address":"467 Hutchinson Court","email":"daleadams@boink.com","city":"Orick","state":"MD"}z

Statement for inserting into employees\_nested:

POST employees\_nested/\_bulk?refresh

{"index":{"\_id":"1"}}

{"id":3,"name":"Bob Smith","title":null,"projects":[{"name":"SQL Spectrum querying","started\_year":1990},{"name":"SQL security","started\_year":1999},{"name":"OpenSearch security","started\_year":2015}]}

{"index":{"\_id":"2"}}

{"id":4,"name":"Susan Smith","title":"Dev Mgr","projects":[]}

{"index":{"\_id":"3"}}

{"id":6,"name":"Jane Smith","title":"Software Eng 2","projects":[{"name":"SQL security","started\_year":1998},{"name":"Hello security","started\_year":2015,"address":[{"city":"Dallas","state":"TX"}]}]}

### Using Inner join

Inner join creates a new result set by combining columns of two indexes based on your join predicates. It iterates the two indexes and compares each document to find the ones that satisfy the join predicates. You can optionally precede the JOIN clause with an INNER keyword. The join predicate(s) is specified by the ON clause.

SQL query:

SELECT

a.account\_number, a.firstname, a.lastname,

e.id, e.name

FROM accounts a

JOIN employees\_nested e

ON a.account\_number = e.id

Result set:

| **a.account\_number** | **a.firstname** | **a.lastname** | **e.id** | **e.name** |
| --- | --- | --- | --- | --- |
| 6 | Hattie | Bond | 6 | Jane Smith |

### Using Cross join

Cross join, also known as cartesian join, combines each document from the first index with each document from the second. The result set is the the cartesian product of documents of both indexes. This operation is similar to the inner join without the ON clause that specifies the join condition.

SQL query:

SELECT

a.account\_number, a.firstname, a.lastname,

e.id, e.name

FROM accounts a

JOIN employees\_nested e

Result set:

| **a.account\_number** | **a.firstname** | **a.lastname** | **e.id** | **e.name** |
| --- | --- | --- | --- | --- |
| 1 | Amber | Duke | 3 | Bob Smith |
| 1 | Amber | Duke | 4 | Susan Smith |
| 1 | Amber | Duke | 6 | Jane Smith |

### Use left outer join Using Left outer join

to retain rows from the first index if it does not satisfy the join predicate. The keyword OUTER is optional.

SQL query:

SELECT

a.account\_number, a.firstname, a.lastname,

e.id, e.name

FROM accounts a

LEFT JOIN employees\_nested e

ON a.account\_number = e.id

Result set:

| **a.account\_number** | **a.firstname** | **a.lastname** | **e.id** | **e.name** |
| --- | --- | --- | --- | --- |
| 1 | Amber | Duke | null | null |
| 6 | Hattie | Bond | 6 | Jane Smith |
| 13 | Nanette | Bates | null | null |
| 18 | Dale | Adams | null | null |

### Using Left outer join

A subquery is a complete SELECT statement used within another statement and enclosed in parenthesis. From the explain output, you can see that some subqueries are actually transformed to an equivalent join query to execute.

### Example 1: Table subquery

SQL query:

SELECT a1.firstname, a1.lastname, a1.balance

FROM accounts a1

WHERE a1.account\_number IN (

SELECT a2.account\_number

FROM accounts a2

WHERE a2.balance > 10000

)

Result set:

| **a1.firstname** | **a1.lastname** | **a1.balance** |
| --- | --- | --- |
| Amber | Duke | 39225 |
| Nanette | Bates | 32838 |

### Example 2: From subquery

SQL query:

SELECT a.f, a.l, a.a

FROM (

SELECT firstname AS f, lastname AS l, age AS a

FROM accounts

WHERE age > 30

) AS a

Step 6: Using SQL Functions

The MATCHQUERY and MATCH\_QUERY functions are synonyms for the [MATCH](https://opensearch.org/docs/latest/search-plugins/sql/full-text#match) relevance function. They don’t accept additional arguments but provide an alternate syntax.

To use matchquery or match\_query, pass in your search query and the field name that you want to search against:

Syntax:

match\_query(field\_expression, query\_expression[, option=<option\_value>]\*)

matchquery(field\_expression, query\_expression[, option=<option\_value>]\*)

field\_expression = match\_query(query\_expression[, option=<option\_value>]\*)

field\_expression = matchquery(query\_expression[, option=<option\_value>]\*)

SELECT account\_number, address

FROM accounts

WHERE MATCHQUERY(address, 'Holmes')

The results contain documents in which the address contains “Holmes”:

| **account\_number** | **address** |
| --- | --- |
| 1 | 880 Holmes Lane |

The QUERY function is a synonym for [QUERY\_STRING](https://opensearch.org/docs/latest/search-plugins/sql/full-text#query-string).

Syntax:

query('query'=query\_expression[, 'fields'=field\_expression][, option=<option\_value>]\*)

The following query function:

SELECT account\_number, address

FROM accounts

WHERE query('address:Lane OR address:Street')

The results contain addresses that contain “Lane” or “Street”:

| **account\_number** | **address** |
| --- | --- |
| 1 | 880 Holmes Lane |
| 6 | 671 Bristol Street |
| 13 | 789 Madison Street |

To return a relevance score along with every matching document, use the SCORE, SCOREQUERY, or SCORE\_QUERY functions.

Syntax:

The SCORE function expects two arguments. The first argument is the [MATCH\_QUERY](https://opensearch.org/docs/latest/search-plugins/sql/sql/functions/#match-query) expression. The second argument is an optional floating-point number to boost the score (the default value is 1.0):

SCORE(match\_query\_expression, score)

SCOREQUERY(match\_query\_expression, score)

SCORE\_QUERY(match\_query\_expression, score)

The following example uses the SCORE function to boost the documents’ scores:

SELECT account\_number, address, \_score

FROM accounts

WHERE SCORE(MATCH\_QUERY(address, 'Lane'), 0.5) OR

SCORE(MATCH\_QUERY(address, 'Street'), 100)

ORDER BY \_score

The results contain matches with corresponding scores:

| **account\_number** | **address** | **score** |
| --- | --- | --- |
| 1 | 880 Holmes Lane | 0.5 |
| 6 | 671 Bristol Street | 100 |
| 13 | 789 Madison Street | 100 |

Another way to query your data with SQL, is to send HTTP requests to \_sql using the following format:

POST *domain-endpoint*/\_plugins/\_sql

{

"query": "SELECT \* FROM my-index LIMIT 50"

}

The Java Database Connectivity (JDBC) driver lets you integrate OpenSearch with your favorite business intelligence (BI) applications. For information about downloading and using the JAR file, see [the SQL repository on GitHub](https://github.com/opensearch-project/sql-jdbc). Installing this will let you setup a connection to OpenSearch from a SQL client such as DBeaver, and you can then run SQL queries from that client. Similarly Amazon Quicksight has a connector to OpenSearch for data visualization. However the most common way for visualizing data in OpenSearch is via OpenSearch dashboards which is a fork from Kibana.

This concludes the SQL tutorial, and you may delete the OpenSearch domain/ index if you are not utilizing it further.