Siren Federate User Guide

Table of Contents

Introduction to Siren Federate

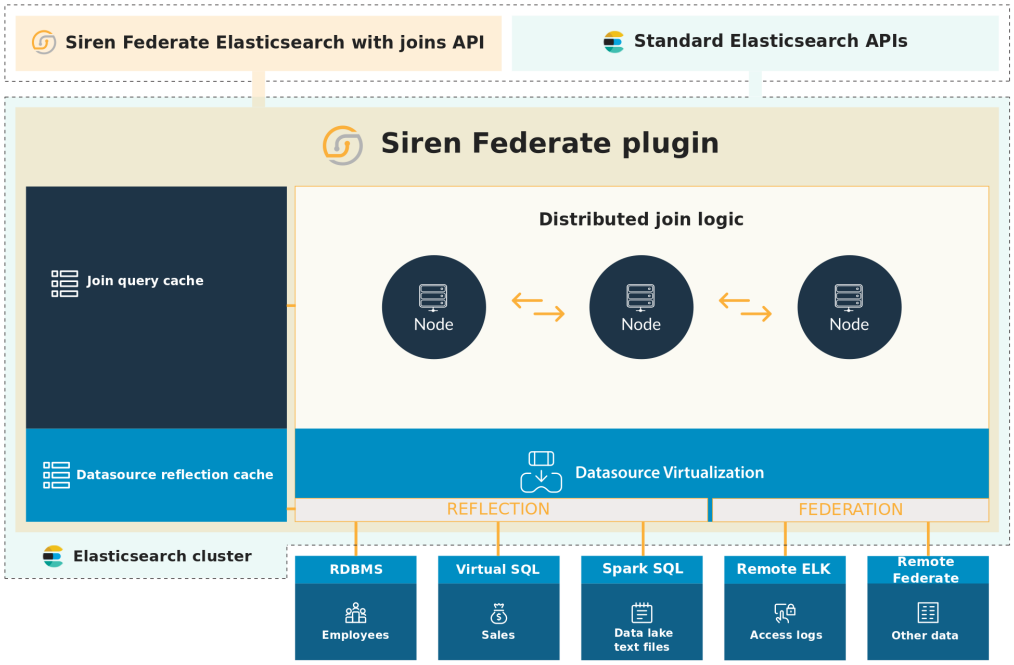
Welcome to the documentation for Siren Federate version {page-component-version}.

NOTE	You can select a previous version by using the dropdown menu in the navigation bar. To access all previous versions, go to www.docs.siren.io.

For a full list of improvements, fixes, and security enhancements, see the release notes .

The Siren Federate plugin extends Elasticsearch with the following main functions:

- A federation layer that enables the virtualization and querying of remote Elasticsearch clusters.
- A reflection layer that enables the caching of data from external databases within Elasticsearch.
- A distributed join layer that enables the execution of join operations at scale.
- A join-caching layer, based on patent-pending techniques, that enables the caching of the most common join results for faster execution times.



The federation of remote Elasticsearch clusters

Siren Federate provides a module, called Connector, which presents indices from remote Elasticsearch clusters as local to the cluster, denoted as 'virtual indices'.

The connector APIs allow you to register a remote Elasticsearch cluster as a datasource.

After a datasource is registered, an index from the remote cluster can be mapped to a virtual index. When a request is sent to the virtual index by using an Elasticsearch API, such as the `Mapping` or `Search` API, the request is intercepted by the Connector module.

The reflection of external databases

Siren Federate provides a feature, called 'Reflection', which enables the import of data into Elasticsearch from an external datasource. A reflection is a recurrent and fully-managed ingestion that replicates the data from a datasource into an Elasticsearch index.

This can be useful in different scenarios. For example, if a user wants to take advantage of the unique search capabilities of the Elasticsearch back-end system, they might want to decrease the load on the external database system, or they might want to increase the performance, given that Elasticsearch is typically faster than SQL back-end systems (such as Spark SQL) for search and analytics.

A distributed join between indices

Siren Federate extends the Elasticsearch Query DSL with a `join` query clause, which enables the execution of a join operation between two sets of documents, based on a join condition. To create complex query plans, you can freely combine and nest multiple join query clauses by using boolean operators, such as conjunctions, disjunctions, or negations.

The join condition is based on an equality operator between two fields and is satisfied when documents have equivalent values for the specified fields. The two fields must be of the same data type. Numerical and textual fields are supported.

Siren Federate currently supports two types of join operation: the (left) semi join and the inner join. The join operation is implemented on top of an in-memory distributed computing layer, which scales with the number of nodes available in the cluster. The join operation is parallelized to scale with the number of CPU cores that are available in a machine.

During the execution of a join operation, projected fields from documents are shuffled across the network and stored in memory. The projected fields are encoded in a columnar format using Apache Arrow and stored in the off-heap memory, therefore reducing its impact on the heap memory.

Semi-join

The semi-join is used to filter one set of documents, A, based on a second set of documents, B. A semi-join between the two sets of documents, A and B, returns the documents of A that satisfy the join condition with the documents of B. This is equivalent to the `EXISTS()` operator in SQL.

Inner join

The inner join enables the “projection” of arbitrary fields (including script fields and document’s scores) from a set of documents, B, and “combines” them with a set of documents, A. The projected fields and associated values of a document from set B are mapped to all of the documents from set A that satisfy the join condition. The result of the join is the set of documents, A, augmented by the projected fields from the set of documents, B.

This inner join is useful when there is a need to materialize a view over many disparate records

located in multiple data sources.

It is common in log analysis, cyber threat inspection, and intelligence investigation to have diverse recorded events about a particular entity, which are spread across multiple data sources. For example, a user can be linked to one or more sessions and a session can be linked to one or more events, such as login, logout, unauthorized actions, and so on. It is difficult to answer questions such as, “find all of the users who were logged in at time t” or “find all of the users who displayed irregular online activity” from a disparate set of records.

In this scenario, the inner join enables the collection and the grouping of multiple events into a particular context for further analysis.

How does the Siren Federate join compare with the Elasticsearch parent-child model?

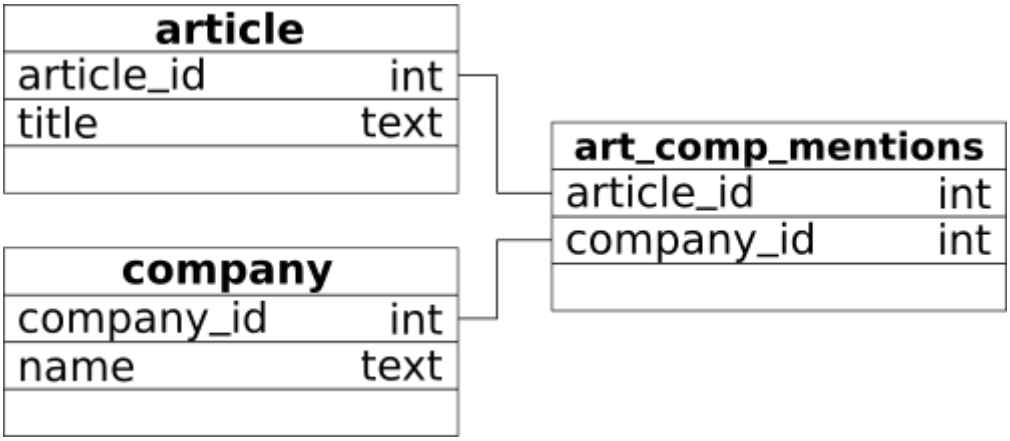
The Siren Federate join is similar in nature to the Parent-Child feature of Elasticsearch: they perform a join at query-time. However, there are important differences between them:

- The parent-child model requires the denormalization of your data model into a hierarchical form. This limits the flexibility of the data modeling and may lead to data redundancy. Siren Federate does not have this data modeling constraint and allows data normalization.
- The parent document and all of its children must live on the same shard. This limits scalability, because you cannot distribute child documents to other shards or, therefore, to other nodes. The Siren Federate join removes this constraint: it allows you to join documents across shards and across indices.
- Thanks to the data locality of the parent-child model, the computation of a join does not require transferring data across the network. On the contrary, Siren Federate needs to transfer data across the network while it computes joins across indices, which impacts its performance.

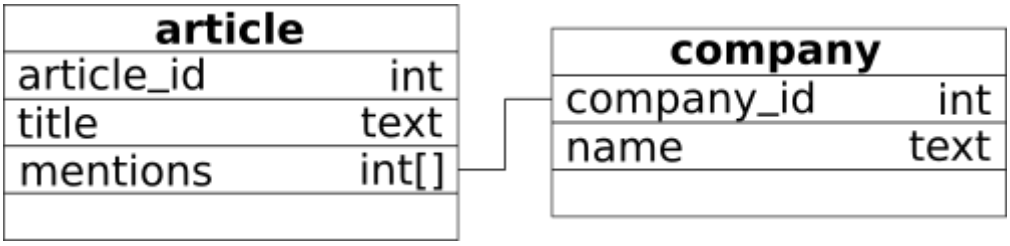
There is no 'one size fits all' solution to this problem, and you need to understand your requirements well to choose the most suitable solution. As a basic rule, if your data model and data relationships are purely hierarchical (or can be mapped to a purely hierarchical model), then the parent-child model might be more appropriate. On the other hand, if you need to query both directions of a data relationship, then the Siren Federate join might be more appropriate.

The data model on which the Siren Federate join operates

The most important requirement for executing a join is to have a common shared attribute between two indices. For example in the diagram below, there is a simple relational data model that is composed of two tables, `article` and `company`, and of one junction table `art_comp_mentions` to encode the many-to-many relationships between them.



This model can be mapped to two Elasticsearch indices, `article` and `company`, as shown in the diagram below. An article document will have a multi-valued field `mentions` with the unique identifiers of the companies mentioned in the article. In other words, the field `mentions` is a foreign key in the `article` table that refers to the primary key of the `company` table.



It is possible and uncomplicated to write an SQL statement to flatten and map relationships into a single multi-valued field. Compared to a traditional database model, where a junction table is necessary, the data model can be simplified by taking advantage of multi-valued fields.

Join query cache

Siren Federate provides a query caching mechanism that enables efficient join processing and reduces the query response time. Query caching exploits the idea of reusing cached query results to answer new queries. Caching not only improves the user's experience, but also reduces the Elasticsearch cluster workload and increases its scalability.

The join query cache in Siren Federate works similarly to the query cache of Elasticsearch. The results of a join query clause, which is a list of document identifiers, are cached efficiently by using a bitset data structure. A semantic definition of the join operation is computed and is used as a signature for the cache entry.

When a new query is received by the system, a signature for each of its join operations is computed and compared with the existing signature that is stored in the cache. The new query can be either totally or partially answered by the cached entries. In the case that it is only partially answered, the query is trimmed based on the cached entries and only the remaining query is executed.

The cache uses an LRU eviction policy: When the cache is full, the least-recently-used query results are evicted to make way for new data. The semantic definition of a join operation captures the lineage and version of its data inputs.

If the data in one of its inputs is modified, then the signature of the join operation will be different.

Therefore, the cache entry that is associated with the previous version of the data inputs will become stale and will be evicted.

If an index has one or more replicas, it is important to specify the `preference` parameter of the search request in order to optimally take advantage of the join query cache.

Architecture

Siren Federate is designed around the following core requirements:

- Low latency, real time interactive response – Siren Federate is designed to power ad hoc interactive, read only queries such as those sent from Siren Investigate.
- Implementation of a fully featured relational algebra, capable of being extended for advanced join conditions, operations and statistical optimizations.
- Flexible in-memory distributed computational framework.
- Horizontal scaling of fully distributed operations, leveraging all the available nodes in the cluster.
- Federated – capable of working on data that spans several Elasticsearch clusters.

Siren Federate is based on the following high level architecture concepts:

- A coordinator node, which is in charge of the query parsing, query planning, and query execution. The Apache Calcite engine creates a logical plan of the query, optimizes the logical plan, and executes a physical plan.
- A set of worker processes, which are in charge of executing the physical operations. Depending on the type of physical operation, a worker process is spawned on a per-node or per-shard basis.
- An in-memory distributed file system that is used by the worker nodes to exchange data, with a compact columnar data representation optimized for analytical data processing, zero copy and zero data serialisation.

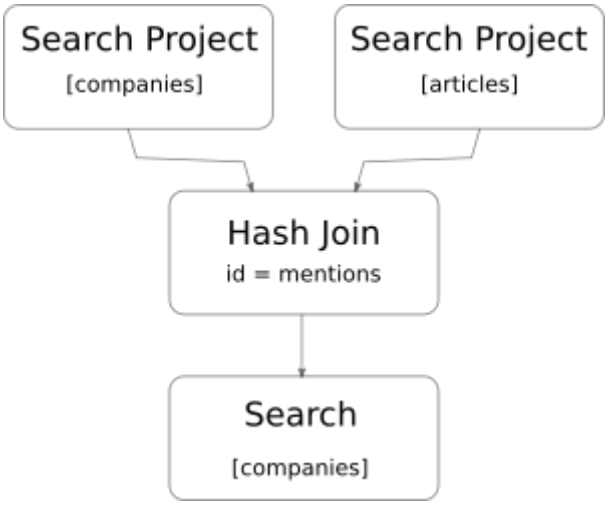
Distributed Join Workflow

When a (multi) search request is sent with one or more nested joins, the node that receives the request becomes the “Coordinator” node. The coordinator node is in charge of controlling and executing a “Job” across the available nodes in the cluster. A job represents the full workflow of the execution of a (multi) search request. A job is composed of one or more “Tasks”. A task represents a single type of operation, such as a `Search/Project` or `Join`, which is executed by a “Worker” on a node. A worker is a thread that performs a task and reports the outcome of the task to the coordinator.

For example, the following search request joining the index `company` with `article`:

```
curl -H 'Content-Type: application/json' -XGET
'http://localhost:9200/siren/company/_search' -d '
{
   "query" : {
      "join" : {
        "type": "HASH_JOIN",
        "indices" : ["article"],
        "on": ["id", "mentions"],
        "request" : {
          "query" : {
            "match_all": {}
          }
        }
      }
    }
}
'
```

will produce the following workflow:



The coordinator executes a `Search/Project` task on every shard of the `company` and `article` indices. These tasks first execute a search query to compute the matching documents, then scan the `id` and `mentions` fields of the matching documents and, finally, shuffle them to all of the nodes of the cluster. Once these tasks are completed, the coordinator executes a `Hash Join` task on every node of the cluster. The `Hash Join` task joins the two streams of data that have been sent by the two previous `Search/Project` tasks to compute a set of document ids for the `company` index. These document identifiers are then transferred back to their respective shards and used to filter the `company` index.

This workflow allows Siren Federate to push all of the filtering predicates (for example, terms, range, or boolean queries) down to Elasticsearch, leveraging the indices for fast computation.

Query Planning & Optimisation

The coordinator node leverages Apache Calcite for planning the job execution. A search request is first parsed into an abstract syntax tree before being transformed into a logical relational plan. A

set of rules will then be applied to optimize the logical plan. We leverage both the Hep and Volcano engines to optimize the logical plan using heuristic and statistical information. The logical plan is then transformed into a physical plan before being executed.

The physical plan represents a tree of tasks to be executed. The coordinator will try to execute tasks concurrently when possible. In the previous example, the two `Search/Project` tasks are executed concurrently, and the `Hash Join` task is executed only after the completion of the two `Search/Project` tasks.

When handling a multi search request, each request will be planned separately, each one producing a physical plan. However, before the execution of the physical plans, the planner will combine all the physical plans into a single one, by mapping identical operations to one single task. We can see that as a step to fold multiple trees of tasks into a single directed graph model, where overlapping operations across trees will become one single vertex in the graph. This is useful to reuse computation across multiple requests.

IO

The shuffling and transfer of data produced by a task is handled by a `Collector`. A collector will collect data, serialize it into a compact columnar data representation, and transfer it in the form of binary packets. Different collector strategies are implemented that are adapted to different tasks. For example, in case of a `Hash Join`, a `Search/Project` task will use a collector with a hash partitioning strategy to create small data partitions and shuffle these partitions uniformly across the cluster.

On the receiver side, when a packet is received, it is stored as is (without deserialization) in an in-memory data store. Tasks, such as the `Join` task, will directly work on top of these binary data packets in order to avoid unnecessary data copy and deserialization.

The binary data packets are created, stored, and manipulated off-heap. This helps to reduce unnecessary loads on the JVM and Garbage Collection when dealing with a large amount of data. Siren Federate leverages the Apache Arrow project for the allocation and management of off-heap byte arrays.

Getting Started

In this short guide, you will learn how you can quickly install the Siren Federate plugin in Elasticsearch, load two sets of documents inter-connected by a common attribute, and execute a relational query across the two sets within the Elasticsearch environment.

Installing the Siren Federate Plugin

From the Elasticsearch installation directory, run the following command:

```
$ ./bin/elasticsearch-plugin install
https://download.support.siren.io/federate/7.16.3-26.5.zip
-> Downloading https://download.support.siren.io/federate/7.16.3-26.5-proguard-
plugin.zip
[=================================================] 100%
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@     WARNING: plugin requires additional permissions     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
* java.io.FilePermission cloudera.properties read
* java.io.FilePermission simba.properties read
* java.lang.RuntimePermission accessClassInPackage.sun.misc
* java.lang.RuntimePermission accessClassInPackage.sun.misc.*
* java.lang.RuntimePermission accessClassInPackage.sun.security.provider
* java.lang.RuntimePermission accessDeclaredMembers
* java.lang.RuntimePermission createClassLoader
* java.lang.RuntimePermission getClassLoader
...
See http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html
for descriptions of what these permissions allow and the associated risks.

Continue with installation? [y/N]y
-> Installed siren-federate
```

To remove the plugin, run the following command:

```
$ bin/elasticsearch-plugin remove siren-federate

-> Removing siren-federate...
Removed siren-federate
```

Starting Elasticsearch

To launch Elasticsearch, run the following command:

```
$ ./bin/elasticsearch
```

In the output, you should see a line like the following which indicates that the Siren Federate plugin is installed and running:

```
[2017-04-11T10:42:02,209][INFO ][o.e.p.PluginsService     ] [etZuTTn] loaded plugin
[siren-federate]
```

Loading Some Relational Data

We will use a simple synthetic dataset for the purpose of this demo. The dataset consists of two sets of documents: Article and Company. An article is connected to a company with the attribute `mentions`. Article will be loaded into the `article` index and company in the `company` index. To load the dataset, run the following command:

```
$ curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/article'
$ curl -H 'Content-Type: application/json' -XPUT
'http://localhost:9200/article/_mapping' -d '
{
  "properties": {
    "mentions": {
      "type": "keyword"
    }
  }
}
'
$ curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/company'
$ curl -H 'Content-Type: application/json' -XPUT
'http://localhost:9200/company/_mapping' -d '
{
  "properties": {
    "id": {
      "type": "keyword"
    }
  }
}
'

$ curl -H 'Content-Type: application/json' -XPUT
'http://localhost:9200/_bulk?pretty&refresh=true' -d '
{ "index" : { "_index" : "article", "_id" : "1" } }
{ "title" : "The NoSQL database glut", "mentions" : ["1", "2"] }
{ "index" : { "_index" : "article", "_id" : "2" } }
{ "title" : "Graph Databases Seen Connecting the Dots", "mentions" : [] }
{ "index" : { "_index" : "article", "_id" : "3" } }
{ "title" : "How to determine which NoSQL DBMS best fits your needs", "mentions" :
["2", "4"] }
{ "index" : { "_index" : "article", "_id" : "4" } }
{ "title" : "MapR ships Apache Drill", "mentions" : ["4"] }

{ "index" : { "_index" : "company", "_id" : "1" } }
{ "id": "1", "name" : "Elastic" }
{ "index" : { "_index" : "company", "_id" : "2" } }
{ "id": "2", "name" : "Orient Technologies" }
{ "index" : { "_index" : "company", "_id" : "3" } }
{ "id": "3", "name" : "Cloudera" }
{ "index" : { "_index" : "company", "_id" : "4" } }
{ "id": "4", "name" : "MapR" }
```

```
'

{
  "took" : 8,
  "errors" : false,
  "items" : [ {
    "index" : {
      "_index" : "article",
      "_id" : "1",
      "_version" : 1,
      "result" : "created",
      "_shards" : {
        "total" : 2,
        "successful" : 2,
        "failed" : 0
      },
      "_seq_no" : 0,
      "_primary_term" : 1,
      "status" : 201
    }
  },
  ...
}
```

Relational Querying of the Data

We will now show you how to execute a relational query across the two indices. For example, we would like to retrieve all the articles that mention companies whose name matches `orient`. This relational query can be decomposed in two search queries: the first one to find all the companies whose name matches `orient`, and a second query to filter out all articles that do not mention a company from the first result set. The Siren Federate plugin introduces a new Elasticsearch filter, named `join`, that allows to define such a query plan and a new search API `siren/<index>/_search` that allows to execute this query plan. Below is the command to run the relational query:

```
$ curl -H 'Content-Type: application/json'
'http://localhost:9200/siren/article/_search?pretty' -d '{ ①
   "query" : {
      "join" : {                      ②
        "indices" : ["company"],     ③
        "on" : ["mentions", "id"],    ④
        "request" : {                 ⑤
          "query" : {
            "term" : {
              "name" : "orient"
            }
          }
        }
      }
    }
}'
```

① The target index (i.e. `article`)

② The `join` query clause

③ The source indices (i.e., `company`)

④ The clause specifying the paths for join keys in both source and target indices

⑤ The search request that will be used to filter out `company` (source set)

The command should return you the following response with two search hits:

```
{
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "article",
      "_id" : "1",
      "_score" : 1.0,
      "_source":{ "title" : "The NoSQL database glut", "mentions" : ["1", "2"] }
    }, {
      "_index" : "article",
      "_id" : "3",
      "_score" : 1.0,
      "_source":{ "title" : "How to determine which NoSQL DBMS best fits your needs",
"mentions" : ["2", "4"] }
    } ]
  }
}
```

You can also reverse the order of the join, and query for all the companies that are mentioned in articles whose title matches `nosql`:

```
$ curl -H 'Content-Type: application/json'
'http://localhost:9200/siren/company/_search?pretty' -d '{
   "query" : {
      "join" : {
        "indices" : ["article"],
        "on": ["id", "mentions"],
        "request" : {
          "query" : {
            "term" : {
              "title" : "nosql"
            }
          }
        }
      }
    }
}'
```

The command should return you the following response with three search hits:

```
{
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "company",
      "_id" : "4",
      "_score" : 1.0,
      "_source":{ "id": "4", "name" : "MapR" }
    }, {
      "_index" : "company",
      "_id" : "1",
      "_score" : 1.0,
      "_source":{ "id": "1", "name" : "Elastic" }
    }, {
      "_index" : "company",
      "_id" : "2",
      "_score" : 1.0,
      "_source":{ "id": "2", "name" : "Orient Technologies" }
    } ]
  }
}
```

Setting up Siren Federate

After Siren Federate is installed, some configuration tasks are required to set it up for your organization's needs.

The following section contains information about connecting datasources, setting up security, configuring join types, and setting limits for memory usage.

Some examples are provided to ensure that you can make the right settings for your needs and get started with Siren Federate as quickly as possible.

In this section

Configuring logging

The default Elasticsearch log configuration can cause an excessive number of log messages when a search request is cancelled.

To reduce the number of messages that are logged, complete the following steps:

1. Open the Log4j 2 properties file `config/log4j2.properties`.
2. Update the value of the `logger.action.level` parameter from `debug` to `warn`.
3. Save and close the file.

For more information about logging in Elasticsearch, see Logging configuration.

Configuring the off-heap memory

To prepare your system for use, you must first configure the off-heap memory.

The memory module is responsible for allocating and managing chunks of off-heap memory. For more information, see the Memory section of the Modules topic.

Checking off-heap memory allocation

Siren Federate provides a cluster-level API that allows you to retrieve statistics about the cluster and the off-heap memory allocation.

The allocated direct memory represents the off-heap memory chunks pre-allocated to accommodate the root allocator.

The chunk of off-heap memory that was allocated is kept and reused, because off-heap memory allocation is expensive.

The root allocator can then allocate off-heap memory buffers of various size in a very efficient way.

Setting off-heap memory

You configure the amount of off-heap memory available for the root allocator by updating the `siren.memory.root.limit` variable in the `config/elasticsearch.yml` file.

To set a limit for the root allocator, you must:

1. Open the `config/elasticsearch.yml` file and set the `siren.memory.root.limit` parameter to a value that is less than the value you set in step 2, for example, 2147483647.

NOTE	It is forbidden to use a limit that is greater than or equal to the maximum direct memory limit.

2. Start the Elasticsearch instance. The following info logs are displayed:

   ```
   [2019-12-10T17:29:11,207][INFO ][i.s.f.c.i.m.BufferAllocatorService] [node_s0]
   Buffer allocator service starting with Unsafe access: true
   [2019-12-10T17:29:11,207][INFO ][i.s.f.c.i.m.BufferAllocatorService] [node_s0]
   Buffer allocator service starting with directMemoryLimit=2147483648 ①
   [2019-12-10T17:29:11,233][INFO ][i.s.f.c.i.m.BufferAllocatorService] [node_s0]
   Buffer allocator service starting with defaultNumDirectArenas=5
   [2019-12-10T17:29:11,236][INFO ][i.s.f.c.i.m.BufferAllocatorService] [node_s0]
   Instantiating root allocator with limit=2147483647 ②
   ```

   These info logs provide an overview of how Siren Federate is configured. Here, we can see that:

   ① The maximum direct memory limit is correctly set to the platform dependent memory limit.

   ② The root allocator limit is correctly set to 2147483647.

Recommended settings

It is critical to ensure that there is enough available memory on the machine to accommodate the maximum direct memory limit for Siren Federate, the JVM maximum heap memory limit, and the operating system.

WARNING	If the sum of maximum direct memory limit for Siren Federate and the JVM maximum heap memory limit does not leave enough memory for the operating system, the OS might stop the Elasticsearch instance (*OOM killer* process on Linux systems).

Configuring a 64GB machine

The following are the recommended settings for a cluster that needs to execute joins on a large amount of data:

- 24 GB heap for Elasticsearch
- 24 GB off-heap for Siren Federate

- 16 GB for the operating system and OS cache

config/jvm.options

```
-Xmx24g
```

config/elasticsearch.yml:

```
siren.memory.root.limit: 25769803775
```

Alternatively, if the off-heap memory for Siren Federate is not fully used, it is better to give more heap memory to Elasticsearch:

- 32 GB heap for Elasticsearch
- 16 GB off-heap for Siren Federate
- 16 GB for the operating system and OS Cache

Configuring a 128GB machine

- 32 GB heap for Elasticsearch
- 64 GB off-heap for Federate
- 32 GB for the operating system and OS Cache

config/jvm.options

```
-Xmx32g
```

config/elasticsearch.yml:

```
siren.memory.root.limit: 68719476735
```

Alternatively, if the off-heap memory for Siren Federate is not fully used, it is better to give more heap memory to the operating system and the OS cache.

Configuring security for Siren Federate

The Siren Federate plugin is compatible with Search Guard and Elastic X-Pack security systems. Follow the instructions to configure one of these solutions for Siren Federate.

A security system maps a user to one or more roles.

A *role* grants one or more permissions, for example, the `sysadmin` role.

A *permission* maps a role to one or more actions.

An *action* specifies a type of request that operates at the index- or cluster level. An action is identified by a unique identifier, for example, `indices:data/read/mget`, which identifies the multi get

action.

An *action* follows the schema `[cluster|indices]:<a path delimited by />`. For example, `cluster:internal/federate/*` or `indices:data/read/mget`.

The following Siren Federate actions can be used to limit cluster or index requests:

- `indices:admin/federate`: The prefix for actions that are related to the administration of internal Siren Federate indices.
- `indices:data/read/federate`: The prefix for actions that are related to reading the index.
- `cluster:monitor/federate`: The prefix for actions that are related to cluster monitoring.
- `cluster:admin/federate`: The prefix for actions that are related to Siren Federate administration.
- `cluster:internal/federate`: The prefix for all internal actions.

NOTE	For every Siren Federate user, you must grant cluster-level permission for internal Siren Federate actions.

Creating roles

To get started, you can create three generic roles in Siren Federate. In a later section, there are examples of how to implement the roles for the different security systems.

The System role

The System role manages the internal Siren Federate indices that store datasource and ingestion configurations.

The following actions can be used for this role:

- `cluster:monitor/*`
- `cluster:admin/*`
- `indices:data/read*`

For datasources and ingestion, the permissions must be granted on the master nodes.

The Admin role

The Admin role performs all actions related to administration, such as managing the license, the datasource, the virtual indices, and the ingestion jobs.

The Admin role grants permissions for Siren Federate cluster administration and monitoring.

To manage, search, and read datasources and virtual indices, grant the following cluster-level and index-level permissions:

- `cluster:admin/federate/connector/*`
- `indices:admin/federate/connector/*`

- `indices:admin/mappings/federate/connector/*`
- `indices:data/read/federate/connector/*`

When a virtual index is defined, Siren Federate ensures that a concrete index exists with the same name for inter-operability with security system plugins. For this reason, you must set the following index-level write permissions:

- `indices:data/write*`

TIP	To simplify the management of virtual indices, use a naming scheme to name them, such as `virtual-*`. You can then grant write permissions to all indices named `virtual-*` collectively.

To manage the license, set the following cluster-level and index-level permissions:

- `cluster:admin/federate/license/*`

To manage the ingestion jobs, set the following permissions:

- `cluster:internal/federate/connector/ingestion/*`
- `cluster:admin/federate/connector/ingestion/*`
- `cluster:admin/ingest/pipeline/put`
- `cluster:admin/ingest/pipeline/delete`
- `indices:admin/create`
- `indices:admin/exists`
- `indices:admin/mapping/put`

The User role

The User role performs read-only actions on indices. This is required to execute a Siren Federate search request on one or more indices (virtual or actual). To search virtual indices, set the following user permissions:

- `indices:data/read/federate/connector/*`

To search indices, set the following user permissions:

- `indices:data/read/*`

These permissions allow the User role to read index metadata. This is required, because the Siren Federate query engine accesses index schema metadata by using `indices:admin/mappings/fields/get` during the query evaluation.

For datasources, the following permissions are required:

- `indices:data/read/federate/connector/*`

Set the following permissions for ingestion jobs:

- `cluster:admin/ingest/pipeline/get`
- `cluster:admin/federate/connector/ingestion/get`
- `cluster:admin/federate/connector/ingestion/run`
- `cluster:admin/federate/connector/ingestion/search`
- `cluster:internal/federate/connector/datasource/get`
- `cluster:internal/federate/connector/ingestion/get`
- `indices:admin/mappings/fields/get`
- `indices:data/read/get`
- `indices:data/read/search`

Securing the connector

After a security system is configured, Siren Federate needs to authenticate as a user with all permissions on the indices that store remote datasource configurations.

The credentials of this user can be specified by setting the following node configurations:

- `siren.connector.username`: The username of the Siren Federate system user.
- `siren.connector.password`: The password of the Siren Federate system user.

Then create a user with that role, for example, a user called `federate_system_user`.

Master node in a cluster with authentication and `federate_system_user`:

elasticsearch.yml

```
siren.connector.username: federate_system_user
siren.connector.password: password
siren.connector.encryption.secret_key: "1zxtIE6/EkAKap+5OsPWRw=="
```

After you set the appropriate configuration parameters, restart the nodes.

NOTE	Write operations are made on the virtual index, not against the actual datasource.

In this section

You can configure one of the following security systems for Siren Federate:

Configuring Search Guard

Configuring Elastic X-Pack

Example of configuring Search Guard

This example implements the generic concepts presented in Configuring security for Siren Federate using Search Guard.

Before you begin, ensure that Search Guard is installed in your cluster, and that you know how to configure users, roles, and permissions.

For more information, see the Search Guard documentation and the introduction in Configuring security for Siren Federate.

Enabling custom headers

Search Guard requires plugins to declare thread headers used. In order for the Federate plugin to work properly, the following node-level setting must be set:

```
searchguard.allow_custom_headers: "_siren_.*"
```

Configuring action groups

The `sg_action_groups.yml` file contains named groups of permissions that can be referred to in the definition of roles. The following are the action groups that are suitable for Siren Federate.

sg_action_groups.yml

```
###### INDEX LEVEL ######

INDICES_ALL:
  allowed_actions:
  - "indices:*"

MANAGE:
  allowed_actions:
  - "indices:monitor/*"
  - "indices:admin/*"

WRITE:
  allowed_actions:
  - "indices:data/write*"
  - "indices:admin/mapping/put"

READ:
  allowed_actions:
  - "indices:data/read*"

VIEW_INDEX_METADATA:
  allowed_actions:
  - "indices:admin/aliases/get"
  - "indices:admin/aliases/exists"
  - "indices:admin/get"
  - "indices:admin/exists"
  - "indices:admin/mappings/fields/get*"
  - "indices:admin/mappings/get*"
  - "indices:admin/mappings/federate/connector/get*"
  - "indices:admin/mappings/federate/connector/fields/get*"
```

```
  - "indices:admin/types/exists"
  - "indices:admin/validate/query"
  - "indices:monitor/settings/get"

###### CLUSTER LEVEL ######

CLUSTER_ALL:
  allowed_actions:
  - "cluster:*"

CLUSTER_MONITOR:
  allowed_actions:
  - "cluster:monitor/*"

CLUSTER_COMPOSITE_OPS:
  allowed_actions:
  -  CLUSTER_COMPOSITE_OPS_RO
  - "indices:data/write/bulk"

CLUSTER_COMPOSITE_OPS_RO:
  allowed_actions:
  - "indices:data/read/mget"
  - "indices:data/read/msearch"
  - "indices:data/read/mtv"
  - "indices:data/read/open_point_in_time"
  - "indices:data/read/close_point_in_time"

CLUSTER_MANAGE:
  allowed_actions:
  - CLUSTER_INTERNAL_FEDERATE
  - "cluster:admin/federate/*"
  - "indices:admin/aliases*"

CLUSTER_INTERNAL_FEDERATE:
  allowed_actions:
  - "cluster:internal/federate/*"
```

Configuring role-based access control

The `sg_roles.yml` file contains a list of user roles. Each role contains a set of permissions at the cluster level and for individual indices.

For example, to define the Admin role and the User role for the `companies` index, open the `sg_roles.yml` file and specify the following:

sg_roles.yml

```
federate_admin:
  cluster_permissions:
    - CLUSTER_MANAGE
    - CLUSTER_MONITOR
  index_permissions:
    - index_patterns:
      - 'companies'
      allowed_actions:
        - MANAGE
        - READ
        - VIEW_INDEX_METADATA

federate_user:
  cluster_permissions:
    - CLUSTER_INTERNAL_FEDERATE
  index_permissions:
    - index_patterns:
      - 'company'
      allowed_actions:
        - READ
        - VIEW_INDEX_METADATA
```

The System role

The following is an example of a System role that can manage internal Siren Federate indices.

sg_roles.yml

```
federate_system:
  index_permissions:
    - index_patterns:
      - '?siren-federate-*'
      allowed_actions:
        - INDICES_ALL
```

The Admin role

The following is an example of an Admin role called `connector_admin` that can manage the index `db_mysql`.

sg_roles.yml

```
connector_admin:
  cluster_permissions:
    - CLUSTER_MANAGE
    - CLUSTER_MONITOR
  index_permissions:
    - index_patterns:
      - 'db_mysql'
      allowed_actions:
        - READ
        - VIEW_INDEX_METADATA
        - MANAGE
```

The User role

The following is an example of a User role called `connector_user` with read-only access to the index called `db_mysql`.

sg_roles.yml

```
connector_user:
  cluster:
    - CLUSTER_INTERNAL_FEDERATE
  index_permissions:
    - index_patterns:
      - 'db_mysql'
      allowed_actions:
        - READ
        - VIEW_INDEX_METADATA
```

The following is an example of a User role called `logs_viewer` that can read-only multiple indices that are prefixed with `logstash-`.

sg_roles.yml

```
logs_viewer:
  index_permissions:
    - index_patterns:
      - 'logstash-*'
      allowed_actions:
        - READ
        - VIEW_INDEX_METADATA
```

Example of configuring Elastic X-Pack

This example implements the generic concepts presented in Configuring security for Siren Federate using Elastic X-Pack.

Before you begin, see the Elastic X-Pack documentation and the introduction on Configuring security for Siren Federate.

Configuring roles

Elastic X-Pack uses roles to define permissions, or `Elastic X-Pack privileges`, on *action* for the cluster and indices. The users are assigned to one of more roles. See the Elastic X-Pack documentation on how to assign users to roles.

The System role

The following is an example of a System role that can manage internal Siren Federate indices.

To configure an Elastic X-Pack system role for your Siren Federate instance, apply the following role settings:

```
{
  "federate_system": {
    "cluster": [
      "cluster:internal/federate/*",
      "cluster:admin/federate/*",
      "cluster:monitor/*"
    ],
    "indices": [
      {
        "names": [
          "/\\.siren.*/"
        ],
        "privileges": [
          "all"
        ]
      },
      {
        "names": [
          "*"
        ],
        "privileges": [
          "indices:monitor/*",
          "indices:admin/*",
          "indices:data/read*",
          "indices:data/write*"
        ]
      }
    ]
  }
}
```

The Admin role

This is an example of an Admin role that can manage the license, datasources, virtual indices, and

the ingestion jobs.

```
{
  "federate_admin": {
    "cluster": [
      "cluster:internal/federate/*",
      "cluster:admin/federate/*",
      "cluster:monitor/*",
      "cluster:admin/xpack/security/*"
    ],
    "indices": [
      {
        "names": [
          "*"
        ],
        "privileges": [
          "indices:monitor/*",
          "indices:admin/*",
          "indices:data/read*"
        ]
      }
    ]
  }
}
```

The User role

This is an example of a User role that has read-only access to indices that are prefixed with `logstash-`.

```
{
  "federate_user": {
    "cluster": [
      "cluster:internal/federate/*"
    ],
    "indices": [
      {
        "names": [
          "logstash-*"
        ],
        "privileges": [
          "indices:data/read*",
          "indices:admin/aliases/get",
          "indices:admin/aliases/exists",
          "indices:admin/get",
          "indices:admin/exists",
          "indices:admin/mappings/fields/get*",
          "indices:admin/mappings/get*",
          "indices:admin/mappings/federate/connector/get*",
          "indices:admin/mappings/federate/connector/fields/get*",
          "indices:admin/types/exists",
          "indices:admin/validate/query",
          "indices:monitor/settings/get",
          "indices:admin/template/get"
        ]
      }
    ]
  }
}
```

The following is an example of a User role called `connector_user` with read-only access to the index called `db_mysql`.

```
{
  "connector_user":{
    "cluster":[
      "cluster:internal/federate/*"
    ],
    "indices":[
      {
        "names":[
          "db_mysql"
        ],
        "privileges":[
          "indices:data/read*",
          "indices:admin/aliases/get",
          "indices:admin/aliases/exists",
          "indices:admin/get",
          "indices:admin/exists",
          "indices:admin/mappings/fields/get*",
          "indices:admin/mappings/get*",
          "indices:admin/mappings/federate/connector/get*",
          "indices:admin/mappings/federate/connector/fields/get*",
          "indices:admin/types/exists",
          "indices:admin/validate/query",
          "indices:monitor/settings/get",
          "indices:admin/template/get"
        ]
      }
    ]
  }
}
```

Connecting to remote datasources

The Siren Federate plugin offers two ways of interacting with data from external sources. It is possible to query external data directly by using *virtual indices*, or indirectly by using *reflections*. A datasource is either an Elasticsearch cluster or a JDBC database.

After a remote datasource is configured, an analyst who is using Siren Investigate can create dashboards by directly querying the remote datasources and displaying the resulting data alongside Elasticsearch data.

A **remote Elasticsearch datasource** causes the system to consider remote indices as local ones; that is, as *virtual indices*.

A **JDBC datasource** allows you to define an external database so that a view of the data can be created by ingesting some of its data into an Elasticsearch index.

Datasources and virtual indices can be managed by using the REST API.

Settings for remote datasources

The Siren Federate plugin stores the datasource configuration in two Elasticsearch indices:

- `.siren-federate-datasources`: The index that is used to store the JDBC configuration parameters of remote datasources.
- `.siren-federate-indices`: The index that is used to store the configuration parameters of virtual indices.

Other indices are also used for different features:

- `.siren-federate-ingestions`: The index that is used to store the ingestion configurations.
- `.siren-federate-joblogs`: The index that is used to store logs of ingestion jobs.

The Siren Federate Connector module supports the datasource node configuration settings. For more information, see the Connector module .

Virtual indices

After you create an Elasticsearch datasource with Siren Federate, you create virtual indices to map external indices.

How do you query virtual indices?

Virtual indices can be queried by using one of the following APIs:

- Standard Elasticsearch API: Allows you to query a virtual index as if it were a standard Elasticsearch index. However, note that this method does not support joins.
- Siren Federate API: Allows you to use the join query clause on virtual indices. The Siren Federate query planner pushes down to the remote datasources the computation of query operators such as filters, aggregations, and, if the Federate plugin is installed on the remote datasource, joins.

Siren Federate supports sophisticated join capabilities across both real indices and virtual indices.

There are three kinds of join operations:

- Joins involving indices within the same external datasource: If the Siren Federate plugin is installed on the datasource, Siren Federate will simply push down the joins to the remote. If the plugin isn't installed, then the `cross back-end join` operation is used. The performance and scalability depends on the datasource that Siren Federate is connected to.
- Cross back-end joins (external datasource to external datasource, external datasource to Elasticsearch): The scalability of this operation is, in its current version, quite high from an external datasource to Elasticsearch, while limited in the opposite direction. Improvements are planned in future versions.
- Joins across indices that are within the same Elasticsearch cluster. These are extremely scalable. Siren Federate augments existing Elasticsearch installations with an in-memory distributed computational layer. Search operations are pushed down to the Elasticsearch indices and then

search results are distributed across the available Elasticsearch nodes for distributed join computation. This enables horizontal scaling, which leverages the entire cluster's CPUs and memory.

Operations on virtual indices

The Siren Federate plugin supports the following operations on virtual indices:

- get mapping
- get field capabilities
- search
- msearch
- get
- mget

NOTE	Search requests that involve a mixture of virtual and normal Elasticsearch indices, for example, when using a wildcard, are not supported and will be rejected. It is, however, possible to issue `msearch` requests that contain requests on normal Elasticsearch indices and virtual indices.

WARNING	Elasticsearch index for interoperability with Search Guard and Elastic X-Pack. If an Elasticsearch index with the same name as the virtual index already exists and it is not empty, the virtual index creation will fail and the original Elasticsearch index is not removed.

Known limitations with configuring remote Elasticsearch datasources

The following limitations exist for all connectors:

- A cross back-end join is limited to semi-join.
- A cross back-end join supports only integer keys.
- Cross back-end support has very different scalability according to the direction of the join. A join that involves sending IDs to a remote system can potentially be hundreds of times less scalable, to one where the keys are fetched from a remote system.
- Cross cluster searches on virtual indices are not supported.

Configuring a remote Elasticsearch connector

Siren Federate provides the capability to query data from an Elasticsearch cluster through the remote clusters module and the Siren Federate connector APIs .

NOTE	The remote Elasticsearch cluster does not have the Siren Federate plugin installed. Therefore Siren Federate cannot push down a join to the remote cluster. Instead, the computation of the join is done on the local cluster using the `broadcast_join` implementation.

Compatibility with security systems

To execute joins spanning several clusters, set the following cluster- and index-level permissions on the clusters.

On the local Federate cluster:

- `cluster:internal/federate/*`
- `indices:data/read/mget`
- `indices:data/read/msearch`
- `indices:data/read/mtv`
- `indices:data/read/open_point_in_time`
- `indices:data/read/close_point_in_time`
- `indices:data/read*`
- `indices:admin/template/get`
- `indices:admin/aliases/get`
- `indices:admin/aliases/exists`
- `indices:admin/get`
- `indices:admin/exists`
- `indices:admin/mappings/fields/get*`
- `indices:admin/mappings/get*`
- `indices:admin/mappings/federate/connector/get*`
- `indices:admin/mappings/federate/connector/fields/get*`
- `indices:admin/types/exists`
- `indices:admin/validate/query`
- `indices:monitor/settings/get`

For the remote ES cluster:

- `indices:data/read/mget`
- `indices:data/read/msearch`
- `indices:data/read/mtv`
- `indices:data/read/open_point_in_time`
- `indices:data/read/close_point_in_time`
- `indices:admin/template/get`
- `indices:data/read*`
- `indices:data/read/search`

The remote Elasticsearch connector is compatible with the following security systems:

- Search Guard
- Elastic X-Pack

Before you begin

1. Ensure that the remote clusters are configured as described in the Configuring remote clusters section of the Elasticsearch documentation.
2. Set up the remote Elasticsearch clusters. For example, use the following settings:

```
curl -X PUT http://localhost:9200/_cluster/settings -H 'Content-type:
application/json' -d '
{
    "persistent": {
        "cluster": {
            "remote": {
                "remotefederate": {
                    "seeds": [
                        "127.0.0.1:9330"
                    ]
                }
            }
        }
    }
}
'
```

Procedure

In this procedure, we are using the example of a remote Elasticsearch cluster called `remoteelasticsearch`, which contains indices called `logs-2019.01`, `logs-2019.02`, ..., `logs-2019.12`, and so on.

1. Define the datasource as an alias to the remote Elasticsearch cluster, by using the Siren Federate datasource API as follows:

```
curl -X PUT http://localhost:9200/_siren/connector/datasource/remoteelasticsearchds
-H 'Content-type: application/json' -d '
  {
    "elastic": {
      "alias": "remoteelasticsearch"
    }
  }
  '
```

2. Define a virtual index on the coordinator cluster that matches the wildcard index pattern `logs-*`, by using the <<modules/siren-federate/pages/connector-apis.adoc#siren-federate-connector-virtual-indices-api,Siren Federate virtual index API> as follows:

```
curl -X PUT http://localhost:9200/_siren/connector/index/logsvi -H 'Content-type:
application/json' -d '
{
  "datasource": "remoteelasticsearchds",
  "resource": "logs-*",
  "key": "_id"
}
'
```

3. Execute a join query. For example, the coordinator cluster contains an index called `machines`, which contains information about IP addresses on machines of interest. To find out about the logs that are associated to these machines, execute the following Federate join query:

```
curl -X GET http://localhost:9200/siren/logsvi/_search -H 'Content-Type:
application/json' -d '
{
    "query": {
        "join": {
            "indices": [
                "machines"
            ],
            "on": [
                "logs_ip_hash",
                "machines_ip_hash"
            ],
            "request": {
                "query": {
                    "match_all": {

                    }
                }
            }
        }
    }
}
'
```

`logs_ip_hash` is the IP field in the index `logsvi` and `machines_ip_hash` is the IP field in the index `machines`.

The API returns the following response:

```
{
  "took": 150,
  "timed_out": false,
  "hits": {
    "total" : {
        "value": 1,
        "relation": "eq"
    },
    "max_score": 1,
    "hits": [
      {
        "_index": "logs-2019-11-12",
        "_id": "0",
        "_score": 2,
        "_source": {
          "date": "2019-11-12T12:12:12",
          "message": "trying out Siren"
        }
      }
    ]
  }
}
```


Known limitations for the Federate connector with a remote Elasticsearch cluster

To use Siren Federate with a remote Elasticsearch cluster, a coordinator Federate cluster must run version 7.11.0.-23.0 or later.

Configuring a remote Federate connector

Siren Federate provides the capability to query data from a Federate cluster. through the remote clusters module, and the Siren Federate connector APIs .

Compatibility with security systems

To execute joins spanning several Federate clusters, set the following cluster- and index-level permissions on the clusters:

- `cluster:internal/federate/*`
- `indices:data/read/mget`
- `indices:data/read/msearch`
- `indices:data/read/mtv`
- `indices:data/read/open_point_in_time`
- `indices:data/read/close_point_in_time`
- `indices:data/read*`
- `indices:admin/template/get`

- `indices:admin/aliases/get`
- `indices:admin/aliases/exists`
- `indices:admin/get`
- `indices:admin/exists`
- `indices:admin/mappings/fields/get*`
- `indices:admin/mappings/get*`
- `indices:admin/mappings/federate/connector/get*`
- `indices:admin/mappings/federate/connector/fields/get*`
- `indices:admin/types/exists`
- `indices:admin/validate/query`
- `indices:monitor/settings/get`

The remote Federate connector is compatible with the following security systems:

- Search Guard
- Elastic X-Pack

Before you begin

1. Ensure that the remote clusters are configured as described in the Configuring remote clusters section of the Elasticsearch documentation.
2. Install the Siren Federate plugin on the remote clusters.
3. Set up the remote Federate clusters. For example, use the following settings:

```
curl -X PUT http://localhost:9200/_cluster/settings -H 'Content-type:
application/json' -d '
{
    "persistent": {
        "cluster": {
            "remote": {
                "remotefederate": {
                    "seeds": [
                        "127.0.0.1:9330"
                    ]
                }
            }
        }
    }
}
'
```

Procedure

In this procedure, we are using the example of a remote Federate cluster called `remotefederate`, which contains indices called `logs-2019.01`, `logs-2019.02`, ..., `logs-2019.12`, and so on.

1. Define the datasource as an alias to the remote Federate cluster, by using the Siren Federate datasource API as follows:

```
curl -X PUT http://localhost:9200/_siren/connector/datasource/remotefederateds -H
'Content-type: application/json' -d '
  {
    "federate": {
      "alias": "remotefederate"
    }
  }
  '
```

2. Define a virtual index on the coordinator cluster that matches the wildcard index pattern `logs-*`, by using the <<modules/siren-federate/pages/connector-apis.adoc#siren-federate-connector-virtual-indices-api,Siren Federate virtual index API> as follows:

```
curl -X PUT http://localhost:9200/_siren/connector/index/logsvi -H 'Content-type:
application/json' -d '
{
  "datasource": "remotefederateds",
  "resource": "logs-*",
  "key": "_id"
}
'
```

3. Execute a join query. For example, the coordinator cluster contains an index called `machines`, which contains information about IP addresses on machines of interest. To find out about the logs that are associated to these machines, execute the following Federate join query:

```
curl -X GET http://localhost:9200/siren/logsvi/_search -H 'Content-Type:
application/json' -d '
{
    "query": {
        "join": {
            "indices": [
                "machines"
            ],
            "on": [
                "logs_ip_hash",
                "machines_ip_hash"
            ],
            "request": {
                "query": {
                    "match_all": {

                    }
                }
            }
        }
    }
}
'
```

`logs_ip_hash` is the IP field in the index `logsvi` and `machines_ip_hash` is the IP field in the index `machines`.

The API returns the following response:

```
{
  "took": 150,
  "timed_out": false,
  "hits": {
    "total" : {
        "value": 1,
        "relation": "eq"
    },
    "max_score": 1,
    "hits": [
      {
        "_index": "logs-2019-11-12",
        "_id": "0",
        "_score": 2,
        "_source": {
          "date": "2019-11-12T12:12:12",
          "message": "trying out Siren"
        }
      }
    ]
  }
}
```


Known limitations for the Federate connector

To use Siren Federate with a remote Federate cluster, a coordinator Federate cluster must run version 7.7.1-20.0 or later, and the remote Federate cluster must run Siren Federate version 7.7.1-20.0 or later.

Troubleshooting datasources

If you experience a problem while configuring remote datasources, see the following information for a possible solution.

I cannot reconnect to a datasource by hostname after a DNS update

When the Java security manager is enabled, the JVM will cache name resolutions indefinitely. If the system that you are connecting to uses round-robin DNS or if the IP address of the system changes frequently, you will need to modify the following Java Security Policy properties:

- `networkaddress.cache.ttl`: The number of seconds to cache a successful DNS lookup. Defaults to `-1` (forever).
- `networkaddress.cache.negative.ttl`: The number of seconds to cache an unsuccessful DNS lookup. Defaults to `10`. Set this value to `0` to avoid caching.

Configuring joins by type

Siren Federate offers three join strategies: the *hash join*, the *broadcast join*, and the *index join*.

By default, the Siren Federate query planner selects the most cost-effective join strategy based on the scenario, but you can also manually select the strategy that you prefer.

About join strategies

- The **hash join** is a fully-distributed join strategy that is designed to join a large number of documents. It scales horizontally (based on the number of data nodes) and vertically (based on the number of CPU cores).
- The **broadcast join** is the strategy to use when you are joining a large set of documents (the parent set) with a small to medium set of documents (the child set).
- The **index join** is the strategy to use when you are joining a large set of documents (the parent set) with a small set of documents (the child set).

When to use a hash join

The hash join is best suited to scenarios where both parent and child sets are large or when the child set is large. A large set is, for example, a set of ten million tuples or more.

The hash join allows the processing of large amounts of data with minimal cost to the network performance.

Therefore, the hash join is useful in a cybersecurity use case, for example, where irregular machine events are being investigated.

NOTE	A tuple is a single row composed of one or more columns, where one column is mapped to one field of a document. For example, a tuple could be a row composed of two elements such as the document identifier and the key value of the join condition. If a document has a multi-valued field, this will generate as many tuples as there are values.

For more term definitions, see the Glossary.

When to use a broadcast join

The broadcast join is best suited to scenarios where the child set of documents (also known as the right-side set) is small or medium (up to a few millions).

Therefore, the broadcast join is useful in a commercial use case, for example, where the sales of a small number of selected products are being observed.

When to use an index join

The index join is best suited to scenarios where the child set of documents (also known as the right-side set) is small (up to a few thousands keys).

Therefore, the index join is useful in a social graph use case, for example, where you start with one person and try to expand his/her relations.

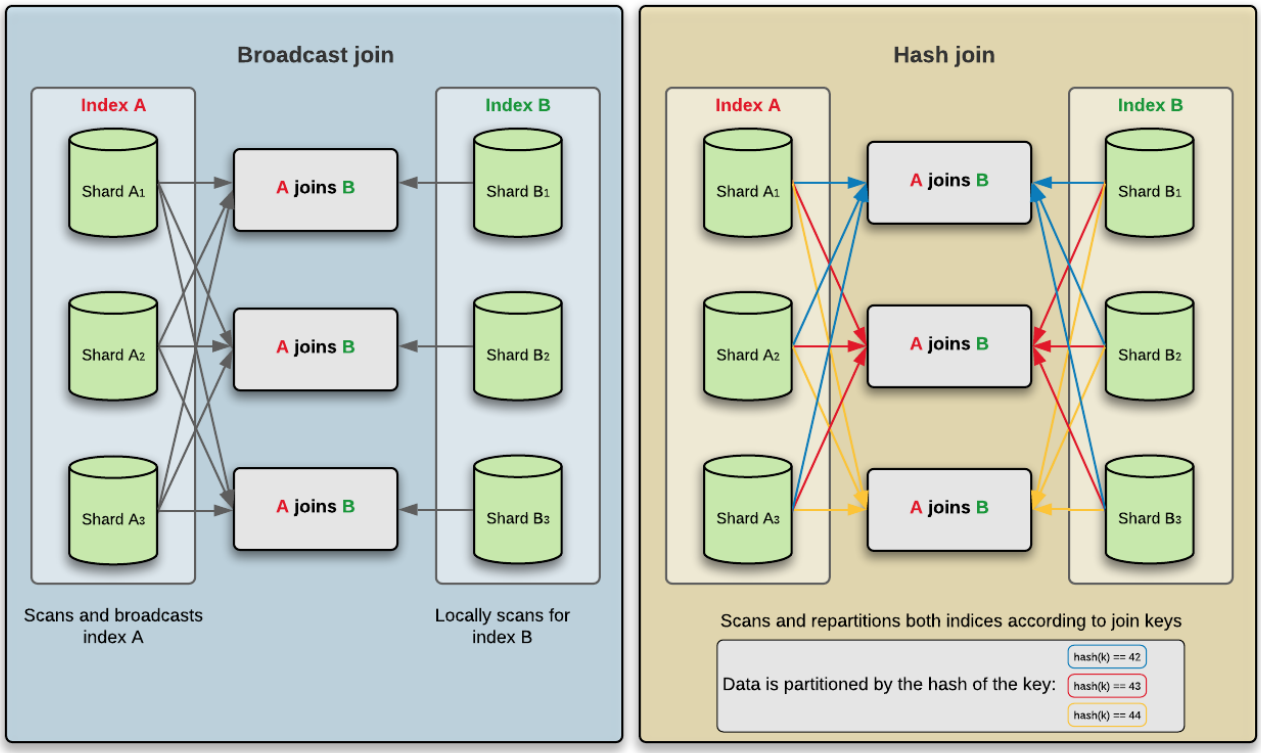
Partitioning data

The difference between the broadcast join and the hash join lies in their approach to data partitioning.

In the following diagram, the broadcast join is shown sending data from the child set of documents (Index B) to every data node in the cluster. Therefore, each data node has the exact same input data as every other data node.

Data from the child set of documents (Index B) is sent only to data nodes that host shards of the parent set of documents (Index A). In addition, if data needs to be sent to two shards that are hosted on the same node, for example, a primary shard and a replica of `Index A`, then the data is sent only once to that node.

In a hash join, data from both sides is partitioned over the key and a data node receives only data with the same hash key.



To summarize, the hash join scales gracefully to process large amounts of data as the number of data nodes increases, thanks to the hash partitioning of the data.

In contrast, the broadcast and index join do not scale well because:

- child data is duplicated across all of the nodes in the cluster, increasing network cost; and
- a receiving node is subject to higher memory cost as the data increases.

Table 1. What happens during the phases of a join?

Join strategy	Shuffle phase (data partitioning)	Build phase	Probe phase
Broadcast join	Copies of the input data (the child set of documents) are sent to every data node.	A in-memory hash table is built over the input data.	The `doc_values` of the parent set of documents are scanned and each value is probed against the hash table to find matches.
Hash join	Data from both sides is partitioned over the key and a data node receives only data with the same hash key.	An in-memory hash table is built from one of the relations in the partition.	The second relation is scanned and each value is probed against the hash table to find matches.
Index join	Copies of the input data (the child set of documents) are sent to every data node.	An in-memory hash table is built over the input data.	Each value is probed against the dictionary of the parent set to find matches.

Impact on performance

With a broadcast or an index join, uploading data to every data node in the cluster has an impact on the network load, the more data and nodes there are. In addition, the memory overhead on a node is linear with the size of the child set. With the broadcast join, `doc_values` of the joined field of the parent set is scanned, while with the index join we perform a dictionary lookup. Indeed, with a small number of terms, the lookup is more efficient.

For this reason, the Siren Query Planner performs a cost analysis to select the more suitable join strategy.

With a hash join, apart from the network load, the fact that the data is partitioned over the cluster also has an impact on the amount of memory needed. However, the created hash table is often smaller.

TIP It is possible to change the parallelization of the hash join computation by using the `siren.io.pipeline.hash.partitions_per_node` setting.

Example of the network, memory, and I/O cost of joins

A three-node cluster contains two indices with fields `fieldA` and `fieldB`, each field containing 15 million values. The field `fieldA` is a field from the parent set, and `fieldB` is a field from the child set.

In addition, each index has three primary shards and no replica. Consequently, one node has one shard of the index.

Hash and broadcast joins are two join strategies that scan the `doc_values` of joined fields. This scan is divided into the three categories that are most impacted: . Network: This exhibits the cost of transferring data between nodes; . I/O: This highlights the cost of reading data from the disk; and .

Memory: This indicates the memory requirements when joining data.

Hash join

To join both indices over fields `fieldA` and `fieldB` with the hash join, the cost would be as follows:

- I/O cost: The system scans `doc_values` of each index before the shuffling phase. This represents a sequential scan of 15M + 15M = 30M values or 30M / 3 = 10M per node.
- Network cost: The system shuffles each index across the available data nodes. This represents a network transfer of 15M + 15M = 30M values across the cluster.
- Memory cost: The system stores the projected data in (off-heap) memory from both child and parent sets. The total number of values stored in memory across the cluster is 15M + 15M = 30M, since there is no duplication when compared to the broadcast join. This represents 30M / 3 = 10M values per node.

Broadcast join

To join both indices over fields `fieldA` and `fieldB` with the broadcast join, the cost would be as follows:

- I/O cost: The system scans `doc_values` of each index and reads 15M + 15M = 30M values from both sides of the join.
- Network cost: The system shuffles the child index across data nodes hosting shards of the parent set. This represents a network transfer of 15M * 3 = 45M values across the cluster.
- Memory cost: The system stores the projected data in (off-heap) memory. The number of values stored in off-heap memory is 15M on each node (values from `fieldB`), which are also loaded into a hash table. This represents 30M on each node, thus a total of 90M values loaded into memory across the cluster.

Index join

To join both indices over fields `fieldA` and `fieldB` with the index join, the cost would be as follows:

- I/O cost: The system scans `doc_values` of the child set and reads 15M values. The system also performs 15M dictionary lookups on the parent index.
- Network cost: The system shuffles the child index across data nodes hosting shards of the parent set. This represents a network transfer of 15M * 3 = 45M values across the cluster.
- Memory cost: The system stores the projected data in (off-heap) memory. The number of values stored in off-heap memory is 15M on each node (values from `fieldB`), which are also loaded into a hash table. This represents 30M on each node, thus a total of 90M values loaded into memory across the cluster.

From this example, we can determine that the hash join strategy is less expensive on the system's operations.

While the broadcast join outshines the hash join when the child index is small (since the parent index is not partitioned over the cluster), in the above case, the broadcast join does not scale very well because:

- the child data is duplicated across all of the nodes in the cluster, increasing network cost; and
- a receiving node is subject to higher memory cost as the data increases.

In contrast, the hash join scales gracefully to process large amounts of data as the number of data nodes increases, thanks to the hash partitioning of the data.

Before you begin

- Firstly, for both the **broadcast** and **hash** join strategies, you must enable `doc_values` for the joined fields. For the **index** join, only the join field of the child set requires `doc_values` to be enabled.

  Fields that have `doc_values` enabled use a columnar data structure for storage, which Siren Federate leverages for efficient scanning.

  If `doc_values` are not enabled, the join fails, stating that the field was not indexed with `doc_values`.

- Secondly, ensure that the data type of the joined fields across index patterns is the same. For example, if you try to join a field from the pattern `index*`, but the field is an **integer** in `index1` while it is a **keyword** in `index2`, an error will result. The following primitive data types are supported:
  - keyword
  - long
  - integer
  - double
  - float
  - short
  - half_float
  - byte
  - date
  - IP
  - binary

Procedure

At query time, specify the join type by entering either `HASH_JOIN`, `BROADCAST_JOIN` or `INDEX_JOIN` in the `type` parameter of the join query.

For more information, see Query DSL.

The search API allows you to execute a search query and get back search hits that match the query. For example, use the following request:

```
` curl -XGET 'http://localhost:9200/siren/<INDEX>/_search' `
```

You can apply a join query and specify the join type that you want to use. For example, the following query specifies a hash join as its type:

```
curl -H 'Content-Type: application/json' -XGET
'http://localhost:9200/siren/target_index/_search' -d '
{
  "query" : {
    "join" : {
      "type": "HASH_JOIN",
      "indices" : ["source_index"],
      "on" : ["foreign_key", "id"],
      "request" : { ①
        "query" : {
          "terms" : {
            "tag" : [ "aaa" ]
          }
        }
      }
    }
  }
}
'
```

① The search request that will be used to filter out the *source* set (that is, the `source_index`).

For more information about the join query, see Query DSL.

Federate Modules

Planner

The planner module is responsible in parsing a (multi) search request and generating a logical model. This logical model is then optimised by leveraging the rule-based Hep engine from Apache Calcite. The outcome is a physical query plan, which is then executed. The physical query plan is a Directed Acyclic Graph workflow composed of individual computing steps. The workflow is executed as a `Job` and the individual computing steps are executed as `Tasks`. We can therefore map one (multi) search request to a single job.

`siren.planner.pool.job.size`

Control the maximum number of concurrent jobs being executed per node. Defaults to 1.

`siren.planner.pool.job.queue_size`

Control the size of the queue for pending jobs per node. Defaults to 100.

`siren.planner.pool.tasks_per_job.size`

Control the maximum number of concurrent tasks being executed per job. Defaults to 3.

`siren.planner.volcano.use_query`

Use contextual queries when computing statistics. If `false`, computed statistics are effectively "global" to the index. Defaults to `false`.

`siren.planner.volcano.cache.enable`

Enable or disable a caching layer over Elasticsearch requests sent during query optimizations in order to gather statistics. Defaults to `true`.

`siren.planner.volcano.cache.refresh_interval`

The minimum interval time for refreshing the cached response of a statistics-gathering request. The time unit is in minutes and defaults to `60` minutes.

`siren.planner.volcano.cache.maximum_size`

The maximum number of requests response that can be cached. Defaults to 1,000,000.

`siren.planner.field.metadata.cache.maximum_size`

The maximum number of field metadata requests response that can be cached. Defaults to 100,000. Setting the value to 0 will disable the cache.

Memory

The memory module is responsible for allocating and managing chunks of off-heap memory.

Memory management

In Siren Federate, data is encoded in a columnar format and stored off-heap. This method of memory management reduces the pressure on the Java virtual machine (JVM) and allows fast and efficient analytical operations.

Data is read directly from the off-heap storage and decoded on-the-fly by using zero-serialization and zero-copy memory. Zero-serialization improves performance by removing any serialization overhead, while zero-copy memory reduces CPU cycles and memory bandwidth overhead.

Siren Federate's memory management allows for granular control over the amount of off-heap memory that can be allocated per node, per search request, and per query operator, while having the inherent ability to terminate queries when the off heap memory usage is reaching its configured limit.

In addition, the garbage collector automatically releases intermediate computation results and recovers the off-heap memory to decrease the impact on memory.

Off-heap storage is used only on the data nodes; master-only and coordinator nodes do not use off-heap memory.

Hierarchical model

The allocated memory is managed in a hierarchical model.

- The `root` allocator is managing the memory allocation on a node level, and can have one or

more `job` allocators.

- A `job` allocator is created for each job (that is, a Siren Federate search request) and manages the memory allocation on a job level. A `job` can have one or more `task` allocators.
- A `task` allocator is created for each task of a job (that is, a Siren Federate query operator) and manages the memory allocation on a task level.

Each allocator specifies a limit for how much off-heap memory it can use.

`siren.memory.root.limit`

Limit in **bytes** for the root allocator. Defaults to two-thirds of the maximum direct memory size of the JVM.

`siren.memory.job.limit`

Limit in **bytes** for the job allocator. Defaults to `siren.memory.root.limit`.

`siren.memory.task.limit`

Limit in **bytes** for the task allocator. Defaults to `siren.memory.job.limit`.

By default, the job limit is equal to the root limit, and the task limit is equal to the job limit. This facilitates a simple configuration in most common scenarios where only the root limit must be configured.

For more advanced scenarios, for example, when there are multiple concurrent users, you might need to tune the job and task limits to avoid errors. For example, a user executes a search request that consumes all of the available off-heap memory at the root level, leaving no memory for the search requests that are executed by other users.

IMPORTANT	As a rule of thumb, do not give more than half of the remaining OS memory to the Siren root allocator. Leave some memory for the OS cache and to cater for Netty's memory management overhead.

For example, if Elasticsearch is configured with a 32GB heap on a machine with 64GB of RAM, this leaves 32GB to the OS. The maximum limit that one could set for the root allocator should be 16GB.

For more information, see Configuring off-heap memory.

IO

The IO module is responsible for encoding, decoding and shuffling data across the nodes in the cluster.

Tuple Collector

This module introduces the concept of `Tuple Collectors` which are responsible for collecting tuples created by a `SearchProject` or `Join` task and shuffling them across the shards or nodes in the cluster.

NOTE	The Tuple Collector is deprecated and will be replaced by the Vectorized Pipeline.

Tuples collected will be transferred in one or more `packets`. The size of a packet has an impact on the resources. Small packets will take less memory but will increase cpu times on the receiver side since it will have to reconstruct a tuple collection from many small packets. Large packets will reduce cpu usage on the receiver side, but at the cost of higher memory usage on the collector side and longer network transfer latency. The size of a packet can be configured with the following setting:

`siren.io.tuple.collector.packet_size`

The size in bytes of a data packet sent by a collector. Defaults to 8MB.

When using the Hash Join, the collector will use a hash partitioner strategy to create small data partitions. Creating multiple small data partitions helps in parallelizing the join computation, as each worker thread for the join task will be able to pick and join one partition independently of the others. Setting the number of data partitions per node to 1 will cancel any parallelization. The number of data partitions per node can be configured with the following setting:

`siren.io.tuple.collector.hash.partitions_per_node`

The number of partitions per node. The number of partitions must be a power of 2. Defaults to 32.

`siren.io.tuple.collector.hash.number_of_nodes`

The number of data nodes that are used during the join computation. This defaults to all available nodes.

Vectorized Pipeline

This module introduces the concept of `Vectorized Pipeline` which is responsible for processing and collecting tuples created by a `SearchProject` or `Join` task and shuffling them across the shards or nodes in the cluster.

Tuples collected will be transferred in one or more `packets`. The size of a packet has an impact on the resources. Small packets will take less memory but will increase cpu times on the receiver side since it will have to reconstruct a tuple collection from many small packets. Large packets will reduce cpu usage on the receiver side, but at the cost of higher memory usage on the collector side and longer network transfer latency. The size of a packet can be configured with the following setting:

`siren.io.pipeline.max_packet_size`

The maximum size in bytes for a data packet. Must be a power of 2. Defaults to 8MB.

`siren.io.pipeline.hash.partitions_per_node`

The number of partitions per node. Must be a power of 2. Defaults to 32.

`siren.io.pipeline.batch_size`

The number of rows in a batch. Must be a power of two. Defaults to 65536.

`siren.io.pipeline.hash.number_of_nodes`

The number of data nodes that are used during the join computation. Defaults to all available nodes.

Thread Pools

Siren Federate introduces new thread pools:

`federate.planner`

For the query planner operations. Thread pool type is `fixed_auto_queue_size` with a size of `2 * # of available_processors`, and initial queue_size of `1000`.

`federate.data`

For the data operations (create, upload, delete). Thread pool type is `scaling`.

`federate.task.worker`

For task worker threads. Thread pool type is `fixed_auto_queue_size` with a size of `max((# of available_processors) - 1, 1)`, and initial queue_size of `1000`.

`federate.connector.query`

For connector query operations. Thread pool type is `fixed` with a size of `int((# of available_processors * 3) / 2) + 1`, and queue size `1000`.

`federate.connector.jobs.management`

For connector job management operations like starting and stopping ingestion jobs. Thread pool type is `scaling`.

`federate.connector.jobs`

For job worker threads like ingestion jobs and related concurrent indexing bulk requests. Thread pool type is `fixed` with a size of `4`, and a queue_size with `100`.

`federate.connector.internal`

For connector internal cluster communications. Thread pool type is `scaling`.

Query Cache

Siren Federate extends the Elasticsearch's query cache:

`index.federate.queries.cache.enabled`

Enable (default) or disable the Siren Federate query cache, used for caching join queries.

`federate.indices.queries.cache.size`

Controls the memory size for the filter cache, defaults to 10%.

`federate.indices.queries.cache.count`

Controls the maximum number of entries in the cache, defaults to 1,000.

Connector

The Federate Connector module supports the following node configuration settings, which can be set on JDBC-enabled nodes:

`siren.connector.datasources.index`

The index in which Federate will store datasource configurations.

`siren.connector.query.project_max_size`

A setting that controls how much data flows between datasources or between a datasource and the Elasticsearch cluster. Defaults to `50000` records transferred between systems consisting in the projected values, e.g., joined values.

`siren.connector.siren.timeout.connection`

the maximum amount of seconds to wait when establishing or acquiring a JDBC connection (`30` by default).

`siren.connector.timeout.query`

the maximum execution time for JDBC queries, in seconds (`30` by default).

`siren.connector.enable_union_aggregations`

`true` by default, can be set to false to disable the use of unions in nested aggregations.

`siren.connector.query.max_bucket_queries`

the maximum number of JDBC queries that will be generated to compute aggregation buckets. Defaults to `500`.

Search APIs

Siren Federate introduces the following new search actions:

- `/siren/<INDEX>/_search` replaces the `/<INDEX>/_search` Elasticsearch action; and
- `/siren/<INDEX>/_msearch` replaces the `/<INDEX>/_msearch` Elasticsearch action.

Both actions are extensions of the original Elasticsearch actions and therefore support the same API.

You must use these actions with the `join` query clause, as the `join` query clause is not supported by the original Elasticsearch actions.

> **Permissions:** To use the APIs that are listed in this section, ensure that the *cluster-level* wildcard action `cluster:internal/federate/*` is granted by the security system.

Search API

The search API allows you to execute a search query and get back search hits that match the query.

Request

```
curl -XGET 'http://localhost:9200/siren/<INDEX>/_search'
```

```
curl -XPOST 'http://localhost:9200/siren/<INDEX>/_search'
```

```
curl -XGET 'http://localhost:9200/siren/_search'
```

```
curl -XPOST 'http://localhost:9200/siren/_search'
```

Path parameter

<index>

(Optional, string) Comma-separated list or wildcard expression of index names used to limit the request.

> **Permissions:** To use this API, ensure that the *index-level* wildcard action `indices:data/read/federate/search*` and the `indices:data/read/federate/planner/search` action are granted by the security system.

Multi Search API

The multi search API allows to execute several search requests within the same API.

Request

```
curl -XGET 'http://localhost:9200/siren/<INDEX>/_msearch'
```

```
curl -XPOST 'http://localhost:9200/siren/<INDEX>/_msearch'
```

```
curl -XGET 'http://localhost:9200/siren/_msearch'
```

```
curl -XPOST 'http://localhost:9200/siren/_msearch'
```

Path parameter

<index>

(Optional, string) Comma-separated list or wildcard expression of index names used to limit the request.

> **Permissions:** To use this API, ensure that the *index-level* wildcard action `indices:data/read/federate/search*` and the `indices:data/read/federate/planner/msearch` action are granted by the security system.

Search Request

The syntax for the body of the search request is identical to the one supported by the Elasticsearch search API, with the additional support for the `join` query clause in the Query DSL.

Parameters

In addition to the parameters supported by the Elasticsearch search API, the Federate search API

introduces the following additional parameters:

`task_timeout` A task timeout, bounding a task to be executed within the specified time value (in milliseconds) and returns with the values accumulated up to that point when expired. Defaults to no timeout (`-1`).

`debug` To retrieve debug information from the query planner. Defaults to `false`.

Taking advantage of the join query cache

The join query cache is responsible for caching the results of a join query clause at the shard level. If an index has one or more replicas, it is recommended that you specify the `preference` parameter of the search request.

If no `preference` parameter is specified, the search request is processed against a random selection of shards. In such a scenario, the join query cache on every shard may differ and the chance of having a positive cache hit decreases.

For example, it is common practice to specify a user session ID as `preference`, so that the same set of shards are selected across the search requests of a same user.

Search Response

The response returned by Federate's search API is similar to the response returned by Elasticsearch's search API. It extends the response with a `planner` object which includes information about the query plan execution.

`is_pruned`

The request response may have been truncated for several reasons and the flag `is_pruned` indicates that the search results are incomplete in the following cases:

- If the `task_timeout` parameter was set.
- If a shard failed.

`query_plan`

If the `debug` parameter is enabled, it will also include detailed information and statistics about the query plan execution within a `query_plan` object.

If the `debug` parameter was disabled and the response was truncated, then a simplified query plan is displayed with information detailing the causes of the truncation.

```
{
    "_shards": {
        "failed": 0,
        "skipped": 0,
        "successful": 5,
        "total": 5
    },
    "hits": {
        "hits": [],
        "max_score": 0.0,
        "total": 0
    },
    "planner": {
        "is_pruned": true,
        "is_truncated": true,
        "node": "AYex2HdPTu-cwkqwaquH1w",
        "query_plan": {
            "children": [
                {
                    "failures": [
                        {
                            "reason": "Unable to allocate buffer of size 2097152 due
to memory limit. Current allocation: 0",
                            "type": "out_of_memory_exception"
                        }
                    ],
                    "type": "SearchTaskBroadcastRequest"
                }
            ],
            "type": "SearchJoinRequest"
        },
        "timestamp": {
            "start_in_millis": 1579776194845,
            "stop_in_millis": 1579776195243,
            "took_in_millis": 398
        },
        "took_in_millis": 398
    },
    "timed_out": false,
    "took": 19
}
```


NOTE The `is_pruned` flag is deprecated and will be renamed to `is_truncated` in version 20.0.

Cancelling a request

A search or a multi search request can be cancelled explicitely by a user. In order to do so, you need

to pass a `X-Opaque-Id` header which is used to identify the request. The endpoint for cancelling a request is `/_siren/job/<ID>/_cancel`. By default, the cancel request will wait for all tasks associated to the search to be cancelled. This can be disabled by passing `false` to the boolean parameter `wait_for_completion`.

> **Permissions:** To use this API, ensure that the *cluster-level* action `cluster:admin/federate/job/cancel` is granted by the security system.

Usage

Let's identify a search request with the name `my-request`:

```
$ curl -H "Content-Type: application/json" -H "X-Opaque-Id: my-request"
'http://localhost:9200/siren/_search'
```

Then to cancel it, issue a request as follows:

```
$ curl -XPOST -H "Content-Type: application/json" 'localhost:9200/_siren/job/my-
request/_cancel'
```

If successful, the response will acknowledge the request and give a listing of the cancelled tasks:

```
{
  "acknowledged" : true,
  "tasks" : [
    {
      "node" : "5ILUA44uSee-VxsBsNbsNA",
      "id" : 947,
      "type" : "transport",
      "action" : "indices:siren/plan",
      "description" : "federate query",
      "start_time_in_millis" : 1524815599457,
      "running_time_in_nanos" : 199131478,
      "cancellable" : true,
      "headers" : {
        "X-Opaque-Id" : "my-request"
      }
    }
  ]
}
```

Validating a request

The `explain` API provides information about the query planning of a search request, without executing it.

Request

```
curl -XGET 'http://localhost:9200/siren/<INDEX>/_explain'
```

```
curl -XPOST 'http://localhost:9200/siren/<INDEX>/_explain'
```

```
curl -XGET 'http://localhost:9200/siren/_explain'
```

```
curl -XPOST 'http://localhost:9200/siren/_explain'
```

Path parameter

`<index>`

(Optional, string) A comma-separated list or a wildcard expression of index names that is used to limit the request.

> **Permissions:** To use this API, ensure that the *index-level* action `indices:data/read/federate/planner/explain` is granted by the security system.

Response

The explain response contains the id of the coordinator node and the physical query plan of the search request.

The query plan is a directed acyclic graph, where each node represents a task that is being executed on the cluster. The graph is represented as a tree to match the JSON data model. Therefore, it might contain duplicate tasks.

Each task node contains the following information:

`type`	Specifies the physical operator type, for example, `SearchJoinRel`, `SearchJoinTaskRel`, or `ParallelHashSemiJoinTaskRel`.
`is_cached`	Indicates whether the physical operator is cached or not.
`request`	Represents the associated search request for a `Search*Rel`; the join condition for a `JoinRel`.
`row_type`	Defines the rows that are being projected by the task. A row is composed of one or more columns. This parameter describes the names and data types of the columns.
`row_count`	An estimation of the number of rows that will be projected.
`cost`	An estimation of the execution cost of the task. This includes the network and I/O costs.

`cumulative_cost` An estimation of the cumulative execution cost of the task. It is the sum of the estimated execution cost of the task and all of its descendants.

When applicable, the `cost` object also details the costs of the different phases; `select` and `project`. This is the case for `SearchJoinRel` and `SearchJoinTaskRel`. For more information about estimating the execution cost, see Example of the network, memory, and I/O cost of joins.

For more information about the workflow phases, see Distributed join workflow.

Example responses

Hash join

```
POST /siren/index1/_explain?pretty=true

{
  "query": {
    "join": {
      "indices": [
        "index2"
      ],
      "type": "HASH_JOIN",
      "on": [
        "foreign_key",
        "id"
      ],
      "request": {
        "query": {
          "bool": {
            "filter": [
              {
                "term": {
                  "tag": {
                    "value": "aaa",
                    "boost": 1
                  }
                }
              }
            ],
            "adjust_pure_negative": true,
            "boost": 1
          }
        }
      }
    }
  }
}
```

```
{
  "node": "RC7OM86mQhGoEW4Q3LVXUg",
  "query_plan": {
    "request": "SearchJoinRequest{jobId=395f99f3-e1c0-43cf-9306-9fbb74c33753,
contextIds=SearchLocks{contextIds=[[RC7OM86mQhGoEW4Q3LVXUg][index1][4]=>[jGiaHnYBx7CeZ
FoovdPz][36], [RC7OM86mQhGoEW4Q3LVXUg][index1][6]=>[jmiaHnYBx7CeZFoovdPz][38],
[RC7OM86mQhGoEW4Q3LVXUg][index1][0]=>[iGiaHnYBx7CeZFoovdPy][32],
[RC7OM86mQhGoEW4Q3LVXUg][index1][3]=>[i2iaHnYBx7CeZFoovdPz][35],
[RC7OM86mQhGoEW4Q3LVXUg][index1][5]=>[jWiaHnYBx7CeZFoovdPz][37],
[RC7OM86mQhGoEW4Q3LVXUg][index1][1]=>[iWiaHnYBx7CeZFoovdPy][33],
[RC7OM86mQhGoEW4Q3LVXUg][index1][2]=>[imiaHnYBx7CeZFoovdPz][34]]},
innerRequest=SearchRequest{searchType=QUERY_THEN_FETCH, indices=[index1],
indicesOptions=IndicesOptions[ignore_unavailable=false, allow_no_indices=true,
expand_wildcards_open=true, expand_wildcards_closed=false,
expand_wildcards_hidden=false, allow_aliases_to_multiple_indices=true,
forbid_closed_indices=true, ignore_aliases=false, ignore_throttled=true], types=[],
routing='null', preference='null', requestCache=false, maxConcurrentShardRequests=5,
batchedReduceSize=512, preFilterShardSize=128, allowPartialSearchResults=null,
localClusterAlias=null, getOrCreateAbsoluteStartMillis=-1, ccsMinimizeRoundtrips=true,
source={\"query\":{\"doc_ids\":{\"job_id\":\"395f99f3-e1c0-43cf-9306-9fbb74c33753\",
\"input_data_id\":\"-1296081227--1507322559-247037071\"}}}}}",
    "row_type": [
      "#0: _shard_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)",
      "#1: _segment_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Short)",
      "#2: _doc_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)",
      "#3: _score JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Float)",
      "#4: foreign_key MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$String) SEARCHABLE AGGREGATABLE}"
    ],
    "type": "SearchJoinRel",
    "physical_plan":
"rel#254:SearchJoinRel.ELASTICSEARCH(input#0=ParallelHashSemiJoinTaskRel#252,invocatio
n=SearchRequest{id=29a689d0-31b1-4163-8b23-
5c92e1af7c9c},rowType=RecordType(JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer) _shard_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Short) _segment_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer) _doc_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Float) _score,
MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$String) SEARCHABLE AGGREGATABLE}
foreign_key),elementType=class [Ljava.lang.Object;)",
    "is_cached": false,
    "row_count": 1,
    "cost": {
      "io": 0,
      "network": 0,
      "project": {
```

```
        "io": 0,
        "network": 0
      },
      "select": {
        "io": 0,
        "network": 0
      }
    },
    "cumulative_cost": {
      "io": 7,
      "network": 12
    },
    "children": [
      {
        "request": "JoinTaskNodesRequest{jobId=395f99f3-e1c0-43cf-9306-9fbb74c33753,
taskType=ParallelHashSemiJoinTask, left_input_data=1323318367--820606795--
8891252244165294113--786400190, right_input_data=3386-316335040--8907078984324448671--
786400190, output_data_id=-1296081227--1507322559-247037071, projection=[1, 2, 3],
condition=(EQUALS, 0, 0), timeout=-1,
collector={class=SegmentPartitionerTupleCollectorManager,
target=[[RC7OM86mQhGoEW4Q3LVXUg][index1][4], [RC7OM86mQhGoEW4Q3LVXUg][index1][6],
[RC7OM86mQhGoEW4Q3LVXUg][index1][0], [RC7OM86mQhGoEW4Q3LVXUg][index1][3],
[RC7OM86mQhGoEW4Q3LVXUg][index1][5], [RC7OM86mQhGoEW4Q3LVXUg][index1][1],
[RC7OM86mQhGoEW4Q3LVXUg][index1][2]]}}",
        "row_type": [
          "#0: _shard_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)",
          "#1: _segment_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Short)",
          "#2: _doc_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)"
        ],
        "type": "ParallelHashSemiJoinTaskRel",
        "physical_plan":
"rel#252:ParallelHashSemiJoinTaskRel.SIREN(left=SearchJoinTaskRel#247,right=SearchJoin
TaskRel#249,condition==($0, $4),joinType=inner)",
        "is_cached": false,
        "row_count": 1.5,
        "cost": {
          "io": 0,
          "network": 5
        },
        "cumulative_cost": {
          "io": 7,
          "network": 12
        },
        "children": [
          {
            "request": "SearchTaskBroadcastRequest{jobId=395f99f3-e1c0-43cf-9306-
9fbb74c33753, taskType=SearchProjectTask, indices=[index1], types=[],
projection=[foreign_key:LONG:class
```

```
io.siren.federate.core.planner.schema.PlannerType$Hashed:false, _shard_id:INT:class
io.siren.federate.core.planner.schema.PlannerType$Integer:false,
_segment_id:SHORT:class io.siren.federate.core.planner.schema.PlannerType$Short:false,
_doc_id:INT:class io.siren.federate.core.planner.schema.PlannerType$Integer:false],
collector={class=HashPartitionerTupleCollectorManager, target=[data:true]}, timeout=-
1, output_data_id=1323318367--820606795--8891252244165294113--786400190,
context_ids=SearchLocks{contextIds=[[RC7OM86mQhGoEW4Q3LVXUg][index1][4]=>[jGiaHnYBx7Ce
ZFoovdPz][36], [RC7OM86mQhGoEW4Q3LVXUg][index1][6]=>[jmiaHnYBx7CeZFoovdPz][38],
[RC7OM86mQhGoEW4Q3LVXUg][index1][0]=>[iGiaHnYBx7CeZFoovdPy][32],
[RC7OM86mQhGoEW4Q3LVXUg][index1][3]=>[i2iaHnYBx7CeZFoovdPz][35],
[RC7OM86mQhGoEW4Q3LVXUg][index1][5]=>[jWiaHnYBx7CeZFoovdPz][37],
[RC7OM86mQhGoEW4Q3LVXUg][index1][1]=>[iWiaHnYBx7CeZFoovdPy][33],
[RC7OM86mQhGoEW4Q3LVXUg][index1][2]=>[imiaHnYBx7CeZFoovdPz][34]]},
input_data_ids=[Lio.siren.federate.core.io.data.DataId;@13bdcd3b, source={\n
\"match_all\" : {\n    \"boost\" : 1.0\n  }\n}}",
            "row_type": [
              "#0: foreign_key MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Hashed) NOT NULL SEARCHABLE
AGGREGATABLE}",
              "#1: _shard_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)",
              "#2: _segment_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Short)",
              "#3: _doc_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)"
            ],
            "type": "SearchJoinTaskRel",
            "physical_plan":
"rel#247:SearchJoinTaskRel.ELASTICSEARCH(invocation=SearchRequest{id=62a60c4d-1c65-
414f-bdd6-4f1fd884cc71},rowType=RecordType(MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Hashed) NOT NULL SEARCHABLE
AGGREGATABLE} foreign_key, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer) _shard_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Short) _segment_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer) _doc_id),elementType=class
[Ljava.lang.Object;)",
            "is_cached": false,
            "row_count": 5,
            "cost": {
              "io": 5,
              "network": 5,
              "project": {
                "io": 5,
                "network": 5
              },
              "select": {
                "io": 0,
                "network": 0
              }
            },
            "cumulative_cost": {
```

```
              "io": 5,
              "network": 5
            }
          },
          {
            "request": "SearchTaskBroadcastRequest{jobId=395f99f3-e1c0-43cf-9306-
9fbb74c33753, taskType=SearchProjectTask, indices=[index2], types=[],
projection=[id:LONG:class
io.siren.federate.core.planner.schema.PlannerType$Hashed:false],
collector={class=HashPartitionerTupleCollectorManager, target=[data:true]}, timeout=-
1, output_data_id=3386-316335040--8907078984324448671--786400190,
context_ids=SearchLocks{contextIds=[[RC7OM86mQhGoEW4Q3LVXUg][index2][4]=>[hWiaHnYBx7Ce
ZFoovdPy][29], [RC7OM86mQhGoEW4Q3LVXUg][index2][2]=>[g2iaHnYBx7CeZFoovdPy][27],
[RC7OM86mQhGoEW4Q3LVXUg][index2][0]=>[gWiaHnYBx7CeZFoovdPy][25],
[RC7OM86mQhGoEW4Q3LVXUg][index2][1]=>[gmiaHnYBx7CeZFoovdPy][26],
[RC7OM86mQhGoEW4Q3LVXUg][index2][5]=>[hmiaHnYBx7CeZFoovdPy][30],
[RC7OM86mQhGoEW4Q3LVXUg][index2][3]=>[hGiaHnYBx7CeZFoovdPy][28],
[RC7OM86mQhGoEW4Q3LVXUg][index2][6]=>[h2iaHnYBx7CeZFoovdPy][31]]},
input_data_ids=[Lio.siren.federate.core.io.data.DataId;@1e7af06, source={\n  \"bool\"
: {\n    \"filter\" : [\n      {\n        \"term\" : {\n          \"tag\" : {\n
\"value\" : \"aaa\",\n            \"boost\" : 1.0\n          }\n        }\n      }\n
],\n    \"adjust_pure_negative\" : true,\n    \"boost\" : 1.0\n  }\n}}",
            "row_type": [
              "#0: id MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Hashed) NOT NULL SEARCHABLE
AGGREGATABLE}"
            ],
            "type": "SearchJoinTaskRel",
            "physical_plan":
"rel#249:SearchJoinTaskRel.ELASTICSEARCH(invocation=SearchRequest{id=4d4024cf-83ec-
4a6a-8d1a-be5b034851fd},rowType=RecordType(MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Hashed) NOT NULL SEARCHABLE
AGGREGATABLE} id),elementType=class [Ljava.lang.Object;)",
            "is_cached": false,
            "row_count": 2,
            "cost": {
              "io": 2,
              "network": 2,
              "project": {
                "io": 2,
                "network": 2
              },
              "select": {
                "io": 0,
                "network": 0
              }
            },
            "cumulative_cost": {
              "io": 2,
              "network": 2
            }
```

```
          }
        ]
      }
    ]
  }
}
```

Broadcast join

```
POST /siren/index1/_explain?pretty=true

{
  "query": {
    "join": {
      "indices": [
        "index2"
      ],
      "type": "BROADCAST_JOIN",
      "on": [
        "foreign_key",
        "id"
      ],
      "request": {
        "query": {
          "bool": {
            "filter": [
              {
                "term": {
                  "tag": {
                    "value": "aaa",
                    "boost": 1
                  }
                }
              }
            ],
            "adjust_pure_negative": true,
            "boost": 1
          }
        }
      }
    }
  }
}
```

```
{
  "node": "nW_8gimES2O-hU0jn3HZBw",
  "query_plan": {
    "request": "SearchJoinRequest{jobId=94662061-49d9-4ac4-bb70-93c2511abffa,
contextIds=SearchLocks{contextIds=[[nW_8gimES2O-
```

```
hU0jn3HZBw][index1][4]=>[YP21HnYBbiKmK-hXe9BC][4], [nW_8gimES2O-
hU0jn3HZBw][index1][2]=>[Xv21HnYBbiKmK-hXe9A-][3], [nW_8gimES2O-
hU0jn3HZBw][index1][6]=>[Yv21HnYBbiKmK-hXe9BN][5],
[RXgnavPjTp6KSZmRzGTdmQ][index1][3]=>[Yf21HnYBbiKmK-hXe9BC][3], [nW_8gimES2O-
hU0jn3HZBw][index1][0]=>[XP21HnYBbiKmK-hXe9A7][2],
[RXgnavPjTp6KSZmRzGTdmQ][index1][1]=>[Xf21HnYBbiKmK-hXe9A-][1],
[RXgnavPjTp6KSZmRzGTdmQ][index1][5]=>[X_21HnYBbiKmK-hXe9BB][2]]},
innerRequest=SearchRequest{searchType=QUERY_THEN_FETCH, indices=[index1],
indicesOptions=IndicesOptions[ignore_unavailable=false, allow_no_indices=true,
expand_wildcards_open=true, expand_wildcards_closed=false,
expand_wildcards_hidden=false, allow_aliases_to_multiple_indices=true,
forbid_closed_indices=true, ignore_aliases=false, ignore_throttled=true], types=[],
routing='null', preference='null', requestCache=false, maxConcurrentShardRequests=5,
batchedReduceSize=512, preFilterShardSize=128, allowPartialSearchResults=null,
localClusterAlias=null, getOrCreateAbsoluteStartMillis=-1, ccsMinimizeRoundtrips=true,
source={\"query\":{\"hash_semi_join\":{\"field\":\"foreign_key\",\"job_id\":\"94662061
-49d9-4ac4-bb70-93c2511abffa\",\"input_data_id\":\"3386-591710918-1684886495832309826-
2147154417\"}}}}}",
    "row_type": [
      "#0: _shard_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)",
      "#1: _segment_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Short)",
      "#2: _doc_id JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer)",
      "#3: _score JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Float)",
      "#4: foreign_key MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$String) SEARCHABLE AGGREGATABLE}"
    ],
    "type": "SearchJoinRel",
    "physical_plan":
"rel#53:SearchJoinRel.ELASTICSEARCH(input#0=SearchJoinTaskRel#48,invocation=SearchRequ
est{id=81d64797-66b7-427a-a7e2-8252d753bf1e},rowType=RecordType(JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer) _shard_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Short) _segment_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Integer) _doc_id, JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Float) _score,
MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$String) SEARCHABLE AGGREGATABLE}
foreign_key),elementType=class [Ljava.lang.Object;)",
    "is_cached": false,
    "row_count": 1,
    "cost": {
      "io": 5,
      "network": 0,
      "project": {
        "io": 0,
        "network": 0
      },
      "select": {
```

```
        "io": 5,
        "network": 0
      }
    },
    "cumulative_cost": {
      "io": 7,
      "network": 4
    },
    "children": [
      {
        "request": "SearchTaskBroadcastRequest{jobId=94662061-49d9-4ac4-bb70-
93c2511abffa, taskType=SearchProjectTask, indices=[index2], types=[],
projection=[id:LONG:class
io.siren.federate.core.planner.schema.PlannerType$Hashed:false],
collector={class=BroadcastTupleCollectorManager, target=[nW_8gimES2O-hU0jn3HZBw,
RXgnavPjTp6KSZmRzGTdmQ]}, timeout=-1, output_data_id=3386-591710918-
1684886495832309826-2147154417, context_ids=SearchLocks{contextIds=[[nW_8gimES2O-
hU0jn3HZBw][index2][0]=>[W_21HnYBbiKmK-hXe9Az][1]]},
input_data_ids=[Lio.siren.federate.core.io.data.DataId;@2a7d114b, source={\n  \"bool\"
: {\n    \"filter\" : [\n      {\n        \"term\" : {\n          \"tag\" : {\n
\"value\" : \"aaa\",\n            \"boost\" : 1.0\n          }\n        }\n      }\n
],\n    \"adjust_pure_negative\" : true,\n    \"boost\" : 1.0\n  }\n}}",
        "row_type": [
          "#0: id MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Hashed) NOT NULL SEARCHABLE
AGGREGATABLE}"
        ],
        "type": "SearchJoinTaskRel",
        "physical_plan":
"rel#48:SearchJoinTaskRel.ELASTICSEARCH(invocation=SearchRequest{id=24aac4ed-e220-
44d8-8803-aa4bac51e0bf},rowType=RecordType(MetadataType{digest=JavaType(class
io.siren.federate.core.planner.schema.PlannerType$Hashed) NOT NULL SEARCHABLE
AGGREGATABLE} id),elementType=class [Ljava.lang.Object;)",
        "is_cached": false,
        "row_count": 2,
        "cost": {
          "io": 2,
          "network": 4,
          "project": {
            "io": 2,
            "network": 4
          },
          "select": {
            "io": 0,
            "network": 0
          }
        },
        "cumulative_cost": {
          "io": 2,
          "network": 4
        }
```

```
      }
    ]
  }
}
```

Query domain-specific language (DSL)

Join query

The `join` filter enables the filtering of one set of documents (the *target*) with another one (the *source*) based on shared field values. It accepts the following parameters:

`type`

The type of join algorithm to use. Valid values are `BROADCAST_JOIN`, `HASH_JOIN`, or `INDEX_JOIN`. If this parameter is not specified, the query planner will automatically select the optimal one. For more information, see Configuring joins by type.

`indices`

The index names for the child set. Multiple indices can be specified using the Elasticsearch syntax. Defaults to all indices.

`on`

An array of two elements that specifies the field paths for the join keys in the parent and the child set, respectively. The metadata of the fields is validated prior to the query execution, see field metadata.

`request`

The search request that is used to compute the set of documents of the child set before performing the join.

Example

In this example, we will join all the documents from `target_index` with the documents of `source_index` using the `HASH_JOIN` algorithm. The query first filters documents from `source_index` and of type `type` with the query `{ "terms" : { "tag" : [ "aaa" ] } }`. It then retrieves the ids of the documents from the field `id` specified by the parameter `on`. The list of ids is then used as filter and applied on the field `foreign_key` of the documents from `target_index`.

```
curl -H 'Content-Type: application/json' -XGET
'http://localhost:9200/siren/target_index/_search' 'd '
{
  "query" : {
    "join" : {
      "type": "HASH_JOIN",
      "indices" : ["source_index"],
      "on" : ["foreign_key", "id"],
      "request" : { ①
        "query" : {
          "terms" : {
            "tag" : [ "aaa" ]
          }
        }
      }
    }
  }
}
'
```

① The search request that will be used to filter out the *source* set (i.e. `source_index`)

Field Metadata

The fields metadata are accessed to check for their capabilities. The field capabilities contain the following information:

`type`
: The data type of the field.

`searchable`
: A property indicating whether or not the field is indexed for search.

`aggregatable`
: A property indicating whether or not the field can be aggregated on.

Both the parent and child fields must be of the same type. For hash and broadcast joins, all join fields must be aggregatable. For index join, parent field must be searchable, and the child field must be aggregatable.

Suppose we have `index1` with two fields, one `keyword` (indexed as searchable and aggregatable) and the other `long` (not indexed as searchable and aggregatable), then we will have the example result below:

```
GET /siren/index1/_field_caps
{
  "indices": [
    "index1"
  ],
  "fields": {
    "comments.author.keyword": {
      "keyword": {
        "type": "keyword",
        "searchable": true,
        "aggregatable": true
      }
    },
    "comments.number": {
      "long": {
        "type": "long",
        "searchable": false,
        "aggregatable": false
      }
    }
  }
}
```

Scoring Capabilities

The `join` filter returns a constant score. Therefore, the scores of the matching documents from the child set do not affect the scores of the matching documents from the parent set. However, one can project the document's score from the child set and customize the scoring of the documents from the parent set with a script score function.

Project

When joining a child set with a parent set, the fields from the child set may be projected to the parent set. The projected fields and associated values are mapped to the matching documents of the parent set.

A projection is defined in the request body search of the join clause using the parameter `project`. The `project` parameter accepts an array of field's objects, each one defining a field to project. There are two different types of field objects: a standard field or a script field.

The projected fields from a child set are accessible in the scope of the parent's request. One can refer to a projected field in a project context or in a script context such as in a script field, a script-based sort, and so on.

Example

Consider the following documents from two indices, `company` and `people`:

```
$ curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/_bulk?pretty'
-d '
{ "index" : { "_index" : "company", "_type" : "company", "_id" : "1" } }
{ "id": 1, "name" : "Acme" }
{ "index" : { "_index" : "company", "_type" : "company", "_id" : "2" } }
{ "id": 2, "name" : "Bueno" }

{ "index" : { "_index" : "people", "_type" : "person", "_id" : "1" } }
{ "id" : 1, "name" : "Alice", "age" : 31, "gender" : "Female", "employed_by" : 1 }
{ "index" : { "_index" : "people", "_type" : "person", "_id" : "2" } }
{ "id" : 2, "name" : "Bob", "age" : 42, "gender" : "Male", "employed_by" : 2 }
{ "index" : { "_index" : "people", "_type" : "person", "_id" : "3" } }
{ "id" : 3, "name" : "Carol", "age" : 26, "gender" : "Female", "employed_by" : 1 }
'
```

Suppose that the two indices are joined in order to retrieve a list of companies with the ages of all their respective employees using the following request:

```
$ curl -H 'Content-Type: application/json'
'http://localhost:9200/siren/company/_search?pretty' -d '{
   "query" : {
      "join" : {
        "indices" : ["people"],
        "on" : ["id", "employed_by"],
        "request" : {
          "project" : [
            { "field" : { "name" : "age", "alias" : "employee_age" } }  ①
          ],
          "query" : {
            "match_all" : {}
          }
        }
      }
    },
    "script_fields" : {
      "employees_age" : {
        "script" : "doc.employee_age"                                   ②
      }
    }
}'
```

① Project the field `age` from index `people` as `employee_age`

② Return a script field `employees_age` for each hit with the associated projected values

The response should contain two hits, one for each company, with the script field `employees_age` as follows:

```
{
  "hits" : {
    "total" : 2,
    "max_score" : 0.0,
    "hits" : [
      {
        "_index" : "company",
        "_type" : "company",
        "_id" : "2",
        "_score" : 0.0,
        "fields" : {
          "employees_age" : [
            42
          ]
        }
      },
      {
        "_index" : "company",
        "_type" : "company",
        "_id" : "1",
        "_score" : 0.0,
        "fields" : {
          "employees_age" : [
            26,
            31
          ]
        }
      }
    ]
  }
}
```


Field

A standard field object specifies the projection of a field from a set. It is composed of the following parameters:

`name`

The name of a field from a child set to project.

`alias`

An alias name to give to the projected field. It is not possible to have multiple fields with identical names in the same set scope as this leads to ambiguity. It is therefore important to carefully select alias names to avoid such ambiguity.

```
{
  "field" : {
    "name" : "age",              ①
    "alias" : "employee_age"     ②
  }
}
```

① The name of the field to project

② An alias for the field name

Script Field

A script field object specifies the projection of the result of a script. It is composed of the following parameters:

`name`

The name given to the projected script field.

`type`

The datatype of the projected script field. Supported datatypes are all numeric datatypes, and keyword datatype.

`script`

The definition of a script supported by the Elasticsearch API. Projected fields from a child set are accessible in the script context using the doc values API.

```
{
  "script_field" : {
    "name" : "employee_age",      ①
    "type" : "integer",           ②
    "script": {                   ③
      "lang": "painless",
      "source": "doc.age"
    }
  }
}
```

① The name of the script field

② The datatype of the script field

③ The script producing values

Document Score

The score of a matching document from a set may be projected using a standard field object using the special field name `_score`.

```
{
  "field" : {
    "name" : "_score",
    "alias" : "employee_score"
  }
}
```

Retrieving a projected field

A script field may be used to retrieve the values of a projected field for each hit, as shown in the previous example. The projected field is accessed using the doc values API. In the example, the projected field `employee_age` is accessed using the syntax `doc.employee_age`.

Sorting based on a projected field

A script-based sorting method can be used to sort the hits based on the values of a projected field, for example:

```
$ curl -H 'Content-Type: application/json'
'http://localhost:9200/siren/company/_search?pretty' -d '{
   "query" : {
     "join" : {
       "indices" : ["people"],
       "on" : ["id", "employed_by"],
       "request" : {
         "project" : [
           {
             "script_field" : {
               "name" : "employee_age",
               "type" : "integer",
               "script" : {
                 "source" : "doc.age",
                 "lang" : "painless"
               }
             }
           }
         ],
         "query" : {
           "match_all" : {}
         }
       }
     }
   },
   "sort": [
    {
      "_script": {
        "script": {
          "lang": "painless",
          "source": "int sum = 0; for (value in doc.employee_age) { sum += value }
return sum;"
        },
        "type": "number",
        "order": "desc"
      }
    }
  ]
}'
```

The response should contain two hits, one for each company, sorted by the sum of their employees age as follows:

```
{
  "hits" : {
    "total" : 2,
    "max_score" : null,
    "hits" : [
      {
        "_index" : "company",
        "_type" : "company",
        "_id" : "1",
        "_score" : null,
        "_source" : {
          "id" : 1,
          "name" : "Acme"
        },
        "sort" : [
          57.0
        ]
      },
      {
        "_index" : "company",
        "_type" : "company",
        "_id" : "2",
        "_score" : null,
        "_source" : {
          "id" : 2,
          "name" : "Bueno"
        },
        "sort" : [
          42.0
        ]
      }
    ]
  }
}
```


Scoring based on a projected field

A script-based scoring method can be used to customize the scoring based on the values of a projected field. For example, we can project the score of the matching documents from the child set and aggregate them into the parent document as follows:

```
$ curl -H 'Content-Type: application/json'
'http://localhost:9200/siren/company/_search?pretty' -d '{
  "query": {
    "function_score": {
      "query": {
        "join": {
          "indices": [ "people" ],
          "on": [ "id", "employed_by" ],
          "request": {
            "project" : [
              { "field" : { "name" : "_score", "alias" : "child_score" } }
            ],
            "query": {
              "match_all": {}
            }
          }
        }
      },
      "functions": [
        {
          "script_score": {
            "script": {
              "lang": "painless",
              "source": "float sum = 0; for (value in doc.child_score) { sum += value
} return sum;"
            }
          }
        }
      ],
      "score_mode": "multiply",
      "boost_mode": "replace"
    }
  }
}'
```

The response should contain two hits, one for each company, sorted by the sum of their child scores as follows:

```
{
  "hits" : {
    "total" : 2,
    "max_score" : 2.0,
    "hits" : [
      {
        "_index" : "company",
        "_type" : "company",
        "_id" : "1",
        "_score" : 2.0,
        "_source" : {
          "id" : 1,
          "name" : "Acme"
        }
      },
      {
        "_index" : "company",
        "_type" : "company",
        "_id" : "2",
        "_score" : 1.0,
        "_source" : {
          "id" : 2,
          "name" : "Bueno"
        }
      }
    ]
  }
}
```


Aggregating based on a projected field

A script can be used to access and aggregate the values of a projected field. For example, we can project the values of the field `gender` of the matching documents from the `people` index and aggregate the documents from the `company` index by using these values as follows:

```
$ curl -H 'Content-Type: application/json'
'http://localhost:9200/siren/company/_search?pretty' -d '{
 "query" : {
    "join" : {
      "indices" : ["people"],
      "on" : ["id", "employed_by"],
      "request" : {
        "project" : [
          { "field" : { "name" : "gender.keyword", "alias" : "employee_gender" } }
        ],
        "query" : {
          "match_all" : {}
        }
      }
    }
  },
  "aggs": {
    "count_by_gender": {
      "terms": {
        "script": {
          "lang": "painless",
          "source": "doc.employee_gender"
        }
      }
    }
  }
}'
```

The response should contain an aggregation result `count_by_gender` with two buckets `Female` and `Male` as follows:

```
{
  "aggregations": {
    "count_by_gender": {
      "doc_count_error_upper_bound": 0,
      "sum_other_doc_count": 0,
      "buckets": [
        {
          "key": "Female",
          "doc_count": 1
        },
        {
          "key": "Male",
          "doc_count": 1
        }
      ]
    }
  }
}
```

Compatibility with Nested Query

The `join` filter within a `nested` query is supported. The join key must specify the field path within the scope of the nested object. For example, as shown below, the join key must be `foreign_key` and not `nested_obj.foreign_key`.

```
curl -H 'Content-Type: application/json' -XGET
'http://localhost:9200/siren/target_index/_search' -d '
{
  "query" : {
    "nested" : {
      "path" : "nested_obj",
      "query" : {
        "join" : {
          "indices" : ["source_index"],
          "on" : ["foreign_key", "id"],
          "request" : {
            "query" : {
              "match_all" : {}
            }
          }
        }
      }
    }
  }
}
'
```

A `nested` query within a `join` filter is also supported if and only if the join key does not refer to a

field of the nested object.

Known limitations

- **A join query involving one or more runtime fields** is currently supported using `HASH_JOIN` or `BROADCAST_JOIN` algorithms. The `INDEX_JOIN` algorithm only supports joins with the runtime field on the child set. Finally, dynamic runtime fields defined in a search request cannot be used in a join query. Runtime fields must be defined in advance in an index mapping.

Paginating a Search Request

A search request can be paginated in Federate using the search_after parameter. The process starts by opening a Point-In-Time (PIT) on the parent indices at the root. This operation creates an identifier that is then passed to the search request to be paginated. This effectively caches the results of the request and ensures consistency of the hits later on. Subsequent pages are then retrieved by re-executing the request and updating the `search_after` parameter. Finally, the PIT must be closed in order to free memory.

Open and Close Point-In-Times

Federate exposes two REST endpoints that allow to open and close Point-In-Times on indices. The state of the indices in the PIT remains unchanged for the duration of the PIT, even if they get updated in the meantime. This allows search requests to be executed against a consistent index, over a long period of time in the midst of potential changes to the indices.

```
POST /siren/<index>/_pit

DELETE /siren/_pit
{
   "id":
"AQ92aV82NzVhOTQ5YV80MWU=#15izAwEWdGVzdHNjcm9sbG5vam9pbi1pbmRleBZtWjNtUVROdlRsT1NkTk9Z
YlVGS1d3ABZoR2FERnR6VlNmbUtPR2RaaXVUVjZnAAAAAAAAAAADFjZJN0t0YTdsUVdtMG95a3pvYjd4NkEAAR
ZtWjNtUVROdlRsT1NkTk9ZYlVGS1d3AAA="
}
```

The POST method opens a PIT on the given index pattern and returns an identifier. The DELETE method closes the PIT referenced by the identifier in its body.

Pagination

Paginating a search request requires the PIT identifier returned by REST API, and a tiebreaker `sort` parameter. The sort parameter is needed to paginate hits: this adds a sort field in the search response that is then passed to the `search_after`. Getting the next page is done by getting the `sort` value of the last returned hit and setting it to the `search_after`.

NOTE	The tiebreaker `sort` parameter is automatically added if there is already a sort in the request.

Below is a search request that contains a join, where the parent set is `machine-*`, and the child set is `beat-*`.

```
GET /siren/machine-*/_search
{
  "query": {
    "join": {
      "indices": [
        "beat-*"
      ],
      "on": [
        "id",
        "machine"
      ],
      "request": {
        "query": {
          "match_all": {}
        }
      }
    }
  }
}
```

A PIT over the parent set at the root is created, i.e., over the index pattern `machine-*`:

```
POST /siren/machine-*/_pit
{
   "id":
"AQ92aV82NzVhOTQ5YV80MWU=#15izAwEWdGVzdHNjcm9sbG5vam9pbi1pbmRleBZtWjNtUVROdlRsT1NkTk9Z
YlVGS1d3ABZoR2FERnR6VlNmbUtPR2RaaXVUVjZnAAAAAAAAAAADFjZJN0t0YTdsUVdtMG95a3pvYjd4NkEAAR
ZtWjNtUVROdlRsT1NkTk9ZYlVGS1d3AAA="
}
```

In order to retrieve the first page, we issue the search request with the identifier and a sort parameter. The index pattern that is normally passed as part of the `_search` endpoint is omitted: indices resolved during the PIT creation are retrieved from the given PIT identifier.

```
GET /siren/_search
{
  "sort": { ①
    "_shard_doc": "asc"
  },
  "pit": { ②
    "id":
"AQ92aV82NzVhOTQ5YV80MWU=#15izAwEWdGVzdHNjcm9sbG5vam9pbi1pbmRleBZtWjNtUVROdlRsT1NkTk9Z
YlVGS1d3ABZoR2FERnR6VlNmbUtPR2RaaXVUVjZnAAAAAAAAAAADFjZJN0t0YTdsUVdtMG95a3pvYjd4NkEAAR
ZtWjNtUVROdlRsT1NkTk9ZYlVGS1d3AAA="
  },
  "query": {
    "join": {
      "indices": [
        "beat-*"
      ],
      "on": [
        "id",
        "machine"
      ],
      "request": {
        "query": {
          "match_all": {}
        }
      }
    }
  },
  "size": 2 ③
}
```

① A sort explicitely set with the tiebreaker field _shard_doc.

② The PIT identifier returned by the call to the `_pit` REST API.

③ The number of hits returned in a page.

In order to retrieve the next pages, the same request must be re-executed, unchanged; the only change is the `search_after` parameter that is added, with the `sort` value from the last returned hit.

```
GET /siren/_search
{
  "sort": {
    "_shard_doc": "asc"
  },
  "pit": {
    "id":
"AQ92aV82NzVhOTQ5YV80MWU=#15izAwEWdGVzdHNjcm9sbG5vam9pbi1pbmRleBZtWjNtUVROdlRsT1NkTk9Z
YlVGS1d3ABZoR2FERnR6VlNmbUtPR2RaaXVUVjZnAAAAAAAAAAADFjZJN0t0YTdsUVdtMG95a3pvYjd4NkEAAR
ZtWjNtUVROdlRsT1NkTk9ZYlVGS1d3AAA="
  },
  "query": {
    "join": {
      "indices": [
        "beat-*"
      ],
      "on": [
        "id",
        "machine"
      ],
      "request": {
        "query": {
          "match_all": {}
        }
      }
    }
  },
  "size": 2,
  "search_after": [ ①
    1
  ]
}
```

① The search_after is given the value of the last returned hit's sort field.

Limitations

The pagination of a search request in Federate currently has the following limitations.

1. The PIT identifier returned by the `/siren/_pit` REST API can only be used by a single search request.
2. A join performed against a virtual indices located on a remote Elasticsearch cluster is not supported, if that remote cluster doesn't have the Federate plugin installed.

Cluster APIs

The cluster APIs enables the retrieval of cluster and node level information, such as statistics about off-heap memory allocation.

Nodes Statistics

The cluster nodes stats API allows to retrieve one or more (or all) of the cluster nodes statistics.

```
curl -XGET 'http://localhost:9200/_siren/nodes/stats'
curl -XGET 'http://localhost:9200/_siren/nodes/nodeId1,nodeId2/stats'
```

The first command retrieves stats of all the nodes in the cluster. The second command selectively retrieves nodes stats of only `nodeId1` and `nodeId2`

By default, all stats are returned. You can limit this by combining any of the following stats:

`memory`
: Memory allocation statistics

`planner`
: Statistics about the planner job and task pools.

Permissions: To use this API, ensure that the *cluster-level* action `cluster:monitor/federate/nodes/stats` is granted by the security system.

Memory Information

The `memory` flag can be set to retrieve information about the memory allocation:

```
curl -XGET 'http://localhost:9200/_siren/nodes/stats/memory'
```

The response includes memory allocation statistics for each node node as follows:

```
{
    "se6baEC9T4K7-14yuG2qgA": {
        "memory": {
            "allocated_direct_memory_in_bytes": 0,
            "allocated_root_memory_in_bytes": 0,
            "root_allocator_dump_reservation_in_bytes": 0,
            "root_allocator_dump_actual_in_bytes": 0,
            "root_allocator_dump_peak_in_bytes": 0,
            "root_allocator_dump_limit_in_bytes": 1073741824
        }
    },
    "sKnVUBo9ShGzkl4GYih7BA": {
        "memory": {
            "allocated_direct_memory_in_bytes": 0,
            "allocated_root_memory_in_bytes": 0,
            "root_allocator_dump_reservation_in_bytes": 0,
            "root_allocator_dump_actual_in_bytes": 0,
            "root_allocator_dump_peak_in_bytes": 0,
            "root_allocator_dump_limit_in_bytes": 1073741824
        }
    }
}
```


`allocated_direct_memory_in_bytes`

The actual direct memory allocated by Netty in bytes

`allocated_root_memory_in_bytes`

The actual direct memory allocated by the root allocator in bytes

`root_allocator_dump_reservation_in_bytes`

Dump of the root allocator initial reservation direct memory allocated.

`root_allocator_dump_actual_in_bytes`

Dump of the root allocator actual direct memory allocated.

`root_allocator_dump_peak_in_bytes`

Dump of the root allocator peak direct memory allocated.

`root_allocator_dump_limit_in_bytes`

Dump of the root allocator limit direct memory allocated.

Planner Information

The `planner` flag can be set to retrieve information about the planner job and task pools:

```
curl -XGET 'http://localhost:9200/_siren/nodes/stats/planner'
```

The response includes memory allocation statistics for each node as follows:


```
{
  "se6baEC9T4K7-14yuG2qgA": {
    "planner": {
      "thread_pool": {
        "job": {
          "permits": 1,
          "queue": 0,
          "active": 0,
          "largest": 1,
          "completed": 538
        },
        "task": {
          "permits": 3,
          "queue": 0,
          "active": 0,
          "largest": 3,
          "completed": 3955
        }
      }
    }
  },
  "sKnVUBo9ShGzkl4GYih7BA": {
    "planner": {
      "thread_pool": {
        "job": {
          "permits": 1,
          "queue": 0,
          "active": 0,
          "largest": 1,
          "completed": 537
        },
        "task": {
          "permits": 3,
          "queue": 0,
          "active": 0,
          "largest": 3,
          "completed": 3863
        }
      }
    }
  }
}
```


Query cache information

To retrieve information about Siren Federate's query cache, you can set the `query_cache` flag, as follows:

```
curl -XGET 'http://localhost:9200/_siren/nodes/stats/query_cache'
```

The response includes statistics about the query_cache on each node:


```
{
  "_nodes": {
    "total": 2,
    "successful": 2,
    "failed": 0
  },
  "cluster_name": "my_cluster",
  "nodes": {
    "tEwWYjpbQzSYghVJVt87QQ": {
      "timestamp": 1545408407569,
      "name": "node_s0",
      "transport_address": "127.0.0.1:41639",
      "host": "127.0.0.1",
      "ip": "127.0.0.1:41639",
      "roles": [
        "master",
        "data",
        "ingest"
      ],
      "query_cache": {
        "memory_size_in_bytes": 0,
        "total_count": 0,
        "hit_count": 0,
        "miss_count": 0,
        "cache_size": 0,
        "cache_count": 0,
        "evictions": 0
      }
    },
    "Dw06QS6oRbS3fEMazn5llQ": {
      "timestamp": 1545408407569,
      "name": "node_s1",
      "transport_address": "127.0.0.1:42841",
      "host": "127.0.0.1",
      "ip": "127.0.0.1:42841",
      "roles": [
        "master",
        "data",
        "ingest"
      ],
      "query_cache": {
        "memory_size_in_bytes": 0,
        "total_count": 0,
        "hit_count": 0,
        "miss_count": 0,
```

```
        "cache_size": 0,
        "cache_count": 0,
        "evictions": 0
      }
    }
  }
}
```

`memory_size_in_bytes`

The size in bytes of the cache

`total_count`

The total number of lookups in the cache

`hit_count`

The number of successful lookups in the cache

`miss_count`

The number of lookups in the cache that failed to retrieve data

`cache_size`

The number of entries in the cache

`cache_count`

The number of entries that have been cached

`evictions`

The number of entries that have been evicted from the cache

Optimizer Statistics Cache

The cluster optimizer cache API allows to retrieve a snaphshot of the query optimizer cache for a list of the cluster nodes.

```
curl -XGET 'http://localhost:9200/_siren/cache'
curl -XGET 'http://localhost:9200/_siren/nodeId1,nodeId2/cache'
curl -XGET 'http://localhost:9200/_siren/cache/clear'
curl -XGET 'http://localhost:9200/_siren/nodeId1,nodeId2/cache/clear'
```

The first command retrieves the state of the optimizer cache for all the nodes in the cluster, while the second only for the desired list of node IDs. The third command invalidates the optimizer cache on every node, while the last command does so for only the selected nodes.

The response includes statistics about the cache use on each node:

```
{
  "aQAf0tIwRtq_n4mBr9SLTw": {
    "size": 92,
    "hit_count": 32,
    "miss_count": 92,
    "eviction_count": 42,
    "load_exception_count": 0,
    "load_success_count": 92,
    "total_load_time_in_millis": 68004
  }
}
```

`size`

The estimated number of entries in the cache.

`hit_count`

The number of cache hits.

`miss_count`

The number of cache misses.

`eviction_count`

The number of evicted entries.

`load_exception_count`

The number of times a request failed to execute as its response was to be put in the cache.

`load_success_count`

The number of times a request was executed successfully as its response was to be put in the cache.

`total_load_time_in_millis`

The time spent in milliseconds to load request responses into the cache.

Permissions: To use this API, ensure that the *cluster-level* action `cluster:monitor/federate/planner/optimizer/stats/get` is granted by the security system.

Index APIs

The index APIs are used to manage individual indices.

Permissions: No specific *index-level* actions are required to use the index APIs. Siren Platform uses the same action as Elasticsearch: `indices:admin/cache/clear`.

Query Cache

Siren's query cache can be cleared together with that of Elasticsearch. For more details, please refer to the Elasticsearch clear cache documentation.

```
curl -XPOST 'http://localhost:9200/<index>/_cache/clear?query=true'
```

The `POST` request clears the query cache for the specified index.

Connector APIs

In this section we present the APIs available to interact with datasources, virtual indices, ingestion jobs.

> **Permissions:** To use the APIs listed in this section, ensure that the *cluster-level* wildcard action `cluster:internal/federate/*` is granted by the security system.

Configuring a JDBC-enabled node

You can ingest data from a JDBC datasource on a node where the Siren Federate plugin is installed.

Before you begin

The Elasticsearch cluster must contain at least one node that is enabled to issue queries over JDBC. It is recommended that you use a coordinating-only node for this role, although this is not a requirement for testing purposes.

To configure the JDBC datasource, you need an Avatica server. The Avatica server in turn connects to the remote JDBC database.

If your system needs additional encryption, generate a custom key by running the `keygen.sh` script.

Procedure

To configure a JDBC datasource, complete the following steps:

1. Open the `elasticsearch.yml` file and add the following setting:

   ```
   node.attr.connector.jdbc: true
   ```

2. Restart the Elasticsearch service.

Configuring encryption for JDBC datasources

JDBC passwords are encrypted by default by using a predefined 128-bit AES key. However, additional encryption is advisable in a production environment.

Before you create datasources, it is recommended that you generate a custom key by running the `keygen.sh` script that is included in the `siren-federate` plugin directory.

Procedure

1. From the `siren-federate` plugin directory, run the following command:

   ```
   bash plugins/siren-federate/tools/keygen.sh -s 128
   ```

   The command outputs a random `base64` key. It is also possible to generate keys longer than 128 bit if your JVM supports it.

2. To use the custom key, set the following parameters in the `elasticsearch.yml` file on master nodes and on all of the JDBC nodes:
   - `siren.connector.encryption.enabled`: `true` by default, but can be set to `false` to disable JDBC password encryption.
   - `siren.connector.encryption.secret_key`: a base64 encoded AES key that is used to encrypt JDBC passwords.

Examples

The following are `elasticsearch.yml` settings that can be used for a master node with a custom encryption key:

```
siren.connector.encryption.secret_key: "1zxtIE6/EkAKap+5OsPWRw=="
```

The following are `elasticsearch.yml` settings for a JDBC node with a custom encryption key:

```
siren.connector.encryption.secret_key: "1zxtIE6/EkAKap+5OsPWRw=="
node.attr.connector.jdbc: true
```

After you save the configuration, restart the nodes to apply the settings.

Datasource API

In this section we present the API available to interact with datasources.

Siren Federate currently supports the following types of datasource:

- **federate** to connect to a remote Federate cluster.
- **Elasticsearch** to connect to a remote Elasticsearch cluster.

- **JDBC** to connect to any external database providing a JDBC driver via an Avatica server.

Datasource management

The endpoint for datasource management is at `/_siren/connector/datasource`.

Datasource creation and modification

A datasource with a specific `id` can be created by issuing a `PUT` request. The body of the request varies with the type of the datasource. A datasource cannot be safely updated by using a `PUT` request due to a lack of concurrency control. By default, it is not allowed to update an existing document.

> **Permissions:** To create a datasource, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/put` is granted by the security system.

Federate datasource

```
curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/_siren/connecto
r/datasource/<id>' -d '
{
  "federate": {
    "alias": "remotename"
  }
}
'
```

Federate configuration parameters:

- `alias`: The name of the configured cluster alias in the remote Federate cluster configuration.

Elasticsearch datasource

```
curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/_siren/connecto
r/datasource/<id>' -d '
{
  "elastic": {
    "alias": "remotename"
  }
}
'
```

Elasticsearch configuration parameters:

- `alias`: The name of the configured cluster alias in the remote cluster configuration.

JDBC datasource

```
curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/_siren/connecto
r/datasource/<id>' -d '
{
  "jdbc": {
    "driver": "org.apache.calcite.avatica.remote.Driver",
    "url": "jdbc:avatica:remote:url=http://localhost:8765;serialization=json",
    "username": "username",
    "password": "password",
  }
}
'
```

JDBC configuration parameters:

- `driver`: the class name of the JDBC driver.
- `url`: the JDBC url of the datasource.
- `username`: the username that will be passed to the JDBC driver when getting a connection (optional).
- `password`: the password that will be passed to the JDBC driver when getting a connection (optional).
- `timezone`: if date and timestamp fields are stored in a timezone other than UTC, specifying this parameter will instruct the plugin to convert dates and times to/from the specified timezone when performing queries and retrieving results.
- `properties`: a map of JDBC properties to be set when initializing a connection.

When updating the datasource, if there was already a password, you have to pass it again. You can either pass the new clear password, or remove the `password` property to keep the previous one.

Datasource retrieval

The datasource configuration can be retrieved by issuing a `GET` request as follows:

```
curl -XGET 'http://localhost:9200/_siren/connector/datasource/<id>'
```

If you want to update the datasource and keep the current password, just remove the "password" property.

> **Permissions:** To allow the retrieval of a datasource, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/get` is granted by the security system.

Datasource deletion

To delete a datasource, issue a `DELETE` request as follows:

```
curl -XDELETE 'http://localhost:9200/_siren/connector/datasource/<id>'
```

Permissions: To allow the deletion of a datasource, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/delete` is granted by the security system.

Datasource listing

To list the datasources configured in the system, issue a `GET` request as follows:

```
curl -XGET 'http://localhost:9200/_siren/connector/datasource/_search?type=federate'
```

Permissions: To allow the listing of datasources, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/search` is granted by the security system.

The parameter `type` is optional and allows to filter datasource results to a specific type: either `federate`, `elastic`, or `jdbc`.

Datasource validation

To validate the connection to a datasource, issue a `POST` request as follows:

```
curl -XPOST 'http://localhost:9200/_siren/connector/datasource/<id>/_validate'
```

Permissions: To allow the validation of a datasource, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/validate` is granted by the security system.

Datasource catalog metadata

To get the metadata related to a datasource connection catalog, issue a `POST` request as follows:

```
curl -XPOST 'http://localhost:9200/_siren/connector/datasource/<id>/_metadata?catalog
=<catalog>&schema=<schema>'
```

The parameters are:

- `catalog`: The name of the catalog,
- `schema`: The name of the schema.

The parameters `catalog` and `schema` are optionals:

- If no catalog parameters is given, the API returns the catalog list.
- If no schema parameters is given, then the catalog parameter must be provided.

The API returns the schema list for the given catalog.

The result is a JSON document which contains the resource list for the given catalog and schema.


```
{
  "_id": "postgres",
  "found": true,
  "catalogs": [
    {
      "name": "connector",
      "schemas": [
        {
          "name": "public",
          "resources": [
            {
              "name": "emojis"
            },
            {
              "name": "Player"
            },
            {
              "name": "Matches"
            },
            {
              "name": "ingestion_testing"
            }
          ]
        }
      ]
    }
  ]
}
```


Permissions: To allow the retrieval of the metadata of a datasource, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/metadata` is granted by the security system.

Datasource field metadata

To get the field metadata related to a datasource connection resource (a table), issue a `POST` request as follows:

```
curl -XPOST 'http://localhost:9200/_siren/connector/datasource/<id>/_resource_metadata?catalog=<catalog>&schema=<schema>&resource=<resource>'
```

The parameters are:

`-catalog`: The name of the catalog, `-schema`: The name of the schema, `-resource`: The name of the resource (table).

The result is a JSON document which contains the columns list for the given catalog, schema and resource. It contains also the name of the primary key if it exists.

```
{
  "_id": "postgres",
  "found": true,
  "columns": [
    {
      "name": "TEAM"
    },
    {
      "name": "ID"
    },
    {
      "name": "NAME"
    },
    {
      "name": "AGE"
    }
  ],
  "single_column_primary_keys": [
    {
      "name": "ID"
    }
  ]
}
```

Permissions: To allow the retrieval of the field metadata of a datasource, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/field-metadata` is granted by the security system.

Datasource transform suggestions

To get a suggestion of a transform configuration that can be used by the ingestion, issue a `POST` request as follows:

```
curl -H 'Content-Type: application/json' -XPOST 'http://localhost:9200/_siren/connect
or/datasource/<id>/_transforms' -d '
{
    "query": "SELECT * FROM events"
}
'
```

It executes the query and returns a collection of transform operations based on the columns returned by the query.

```
{
  "_id": "postgres",
  "found": true,
  "transforms": [
    {
      "input": [
        {
          "source": "id"
        }
      ],
      "output": "id"
    },
    {
      "input": [
        {
          "source": "occurred"
        }
      ],
      "output": "occurred"
    },
    {
      "input": [
        {
          "source": "value"
        }
      ],
      "output": "value"
    },
    {
      "input": [
        {
          "source": "location"
        }
      ],
      "output": "location"
    }
  ]
}
```

Permissions: To suggest a transformation, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/datasource/suggest/transform` is granted by the security system.

Virtual index API

In this section we present the API available to interact with the virtual indices.

Virtual index management

Virtual index creation and modification

A virtual index with a specific `id` can be updated by issuing a `PUT` request as follows:

```
curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/_siren/connecto
r/index/<id>'  -d '
{
  "datasource": "ds",
  "catalog": "catalog",
  "schema": "schema",
  "resource": "table",
  "key": "id",
  "search_fields": [
    {
     "function": "LIKE",
     "field": "NAME"
    }
  ]
}
'
```

The id of a virtual index must be a valid lowercase Elasticsearch index name; it is recommended to start virtual indices with a common prefix to simplify handling of permissions.

Body parameters:

- `datasource`: the id of an existing Elasticsearch datasource.
- `resource`: the name of a table or view on the remote datasource.
- `key`: the name of a unique column; if a virtual index has no primary key it will be possible to perform aggregations, however queries that expect a reproducible unique identifier will not be possible.
- `catalog` and `schema`: the catalog and schema containing the table specified in the `resource` parameter; these are usually required only if the connection does not specify a default catalog or schema.
- `search_fields`: An optional list of field names that will be searched using the LIKE operator

when processing queries. Currently only the LIKE function is supported.

Permissions: To create a virtual index, ensure that the *index-level* action `indices:admin/federate/connector/put` is granted by the security system.

Whenever a virtual index is created, the Siren Federate plugin creates a concrete Elasticsearch index with the same name as the virtual index, which would contain some metadata about the virtual index. When starting up, the Siren Federate plugin will check for missing concrete indices and will attempt to create them automatically. For more information, see Operations on virtual indices.

Virtual index deletion

To delete a virtual index, issue a `DELETE` request as follows:

```
curl -XDELETE 'http://localhost:9200/_siren/connector/index/<id>'
```

When a virtual index is deleted, the corresponding concrete index is not deleted automatically.

Permissions: To delete a virtual index, ensure that the *index-level* action `indices:admin/federate/connector/delete` is granted by the security system.

Virtual index listing

To list the virtual indices configured in the system, issue a `GET` request as follows:

```
curl -XGET 'http://localhost:9200/_siren/connector/index/_search'
```

Permissions: To list virtual indices, ensure that the *index-level* action `indices:admin/federate/connector/search` is granted by the security system.

Ingestion API

Permissions: To use the ingestion API, you must ensure that the following *cluster-level* actions are granted by the security system:

- Delete: `cluster:admin/federate/connector/ingestion/delete`
- Encryption: `cluster:admin/federate/connector/ingestion/encrypt`
- Get: `cluster:admin/federate/connector/ingestion/get`
- Put: `cluster:admin/federate/connector/ingestion/put`
- Run: `cluster:admin/federate/connector/ingestion/run`
- Search: `cluster:admin/federate/connector/ingestion/search`
- Validate: `cluster:admin/federate/connector/ingestion/validate`

Ingestion management

The endpoint for ingestion management is at `/_siren/connector/ingestion`.

Datasource query sample

This method runs a query and returns an array of results and an Elasticsearch type for each column found.

```
curl -H 'Content-Type: application/json' -XPOST 'http://localhost:9200/_siren/connect
or/ingestion/<id>/_sample' -d '
{
  "query": "SELECT * FROM events",
  "row_limit": 10,
  "max_text_size": 100
}
'
```

```
{
  "_id": "valid",
  "found": true,
  "types": {
    "location": "keyword",
    "id": "long",
    "occurred": "date",
    "value": "long"
  },
  "results": [
    {
      "id": 0,
      "occurred": 1422806400000,
      "value": 1,
      "location": "Manila"
    },
    {
      "id": 1,
      "occurred": 1422806460000,
      "value": 5,
      "location": "Los Angeles"
    },
    {
      "id": 2,
      "occurred": 1422806520000,
      "value": 10,
      "location": "Pompilio"
    }
  ]
}
```

Permissions: To sample a datasource, you must ensure that the *cluster-level* action `cluster:admin/federate/connector/ingestion/sample` is granted by the security system.

Ingestion creation and modification

An ingestion with a specific `id` can be updated by issuing a `PUT` request as follows. An ingestion with a specific `id` can be created by issuing a `PUT` request. An ingestion can be safely updated by a `PUT` request due to the implementation of `_seq_no` and `_primary_term` fields which enables concurrent modification.

```
curl -H 'Content-Type: application/json' -XPUT 'http://localhost:9200/_siren/connecto
r/ingestion/<id>' -d '
{
  "ingest": {
    "datasource": "postgres",
    "query": "select * from events {{#max_primary_key}}WHERE
```

```
id>{{max_primary_key}}{{/max_primary_key}} limit 10000",
    "batch_size": 10,
    "schedule": "0 0 * * * ?",
    "enable_scheduler": true,
    "target": "events",
    "staging_prefix": "staging-index",
    "strategy": "REPLACE",
    "pk_field": "id",
    "mapping": {
      "properties": {
        "id": { "type": "long" },
        "value": { "type": "keyword" },
        "location": { "type": "text" },
        "geolocation": { "type": "geo_point" }
        }
    },
    "pipeline": {
      "processors": [
          {
            "set" : {
              "field": "foo",
              "value": "bar"
            }
          }
      ]
    },
    "transforms": [{
      "input": [{"source": "id"}],
      "output": "id",
       "mapping": {
        "type": "long"
      }
    },{
      "input": [
        {"source": "lat"},
        {"source": "lon"}
      ],
      "output": "geolocation",
      "transform": "geo_point",
      "mapping": {
        "type": "geo_point"
      }
    }],
    "ds_credentials": {
      "username": "user",
      "password": "pass"
    },
    "es_credentials": {
       "username": "user",
       "password": "pass"
    },
```

```
    "description": "description"
  }
}
'
```

Body parameters:

- `ingest`: the properties of the ingestion.

Ingest configuration parameters:

- `datasource`: the name of a datasource.
- `query`: the template query passed to the JDBC driver collecting the record to ingest.
- `batch_size`: An optional batch size (overriding the default global value).
- `schedule`: An optional schedule using the cron syntax.
- `enable_schedule`: enable or disable the scheduled execution.
- `target`: A target Elasticsearch index name.
- `staging_prefix`: An optional prefix for the staging Elasticsearch index.
- `strategy`: An update strategy. It can be either INCREMENTAL or REPLACE.
- `pk_field`: A primary key field name.
- `mapping`: An Elasticsearch mapping definition.
- `pipeline`: An optional pipeline configuration.
- `transforms`: A sequence of transforms to map the fields declared by the query to the fields in the Elasticsearch mapping definition.
- `ds_credentials`: An optional set of credentials used to connect to the database.
- `es_credentials`: The optional credentials that will be used to perform Elasticsearch requests during jobs.
- `description`: An optional description.

Strategy:

There are two available ingestion strategies:

- `INCREMENTAL`: The index is created if it does not exist. The ingested records are inserted or updated in place.
- `REPLACE`: The index name is an alias to a staging index. The ingested records are inserted on the staging index. When the ingestion is done the alias is moved from the previous staging index to the new one. If anything wrong happens the alias is untouched and points to the previous staging index.

Ingestion query:

The query defined in the ingestion configuration is written in the datasource language. The query can be written using mustache and the following variables are provided, if applicable, when

converting the query to a string:

- `max_primary_key`: the maximum value of the primary key in Elasticsearch.
- `last_record_timestamp`: the UTC timestamp at which the last record was successfully processed by an ingestion job.
- `last_record`: an object with the scalar values in the last record that was successfully processed by the ingestion job.

Mapping transform:

A mapping transform takes one or more fields from a datasource record as inputs and outputs a field that can be indexed with a valid Elasticsearch type.

A mapping transform has the following properties:

- `input`: a sequence of inputs, where an input can be either the name of a field defined in the job query or the name of a field in the target Elasticsearch mapping.
- `transform`: the name of a predefined function that takes as input the values of the fields specified in the input parameter and the mapping properties of the target Elasticsearch field. The function outputs the value to be indexed; if transform is not set, the system uses a generic cast function to create the output.
- `output`: the name of the target Elasticsearch field.

Input:

The input structure must provide one of the following properties:

- `source`: the name of a field defined in the job query.
- `target`: the name of a field in the target Elasticsearch mapping.

Transforms (“predefined functions”):

- `copy`: a default cast transform that produces a scalar Elasticsearch value in a way analogous to how the connector already translates JDBC types to Elasticsearch types. If the JDBC driver reports array fields / struct fields correctly, they will be written as Elasticsearch arrays. Any JDBC type that is not supported / not recognized causes an exception.
- `geo_point`: transform that produces a geo_point value from two numerical inputs, where the first is the latitude and the second the longitude.
- `array`: an array transform that produces an array Elasticsearch value from a comma separated string field in a record.

Credential parameters (for ElasticSearch or the JDBC database):

If the user does not have the permission to manage datasources in the cluster these credentials are mandatory.

- `username`: the login to use to connect to the resource.
- `password`: the password to use to connect to the resource.

When updating the ingestion properties, if there was already a password, you have to pass it again. You can either pass the new clear password, or remove the `password` property to keep the previous one.

Ingestion retrieval

The ingestion properties can be retrieved by issuing a `GET` request as follows:

```
curl -XGET 'http://localhost:9200/_siren/connector/ingestion/<id>'
```

Ingestion deletion

To delete an ingestion, issue a `DELETE` request as follows:

```
curl -XDELETE 'http://localhost:9200/_siren/connector/ingestion/<id>'
```

Ingestion listing

To list the ingestions configured in the system, issue a `GET` request as follows:

```
curl -XGET 'http://localhost:9200/_siren/connector/ingestion/_all?status=[false|true]
&detailed=[false|true]'
```

NOTE `curl -XGET 'http://localhost:9200/_siren/connector/ingestion/_search'` API has been deprecated and is scheduled to be removed in next major release.

If the optional status parameter is set to true, it also returns the last job status, and the last job log.

If the optional detailed parameter(true by default) is set to false, then `mapping, pipeline, transforms and removed_fields` are not returned.

Ingestion validation

To validate the connection to an ingestion, issue a `POST` request as follows:

```
curl -XPOST 'http://localhost:9200/_siren/connector/ingestion/<id>/_validate'
```

Run an ingestion job

To execute an ingestion job, issue a `POST` request as follows:

```
curl -XPOST 'http://localhost:9200/_siren/connector/ingestion/<id>/_run'
```

The response returns the jobId that can be use to track the status of the running job:

```
{
  "_id": "postgres-events",
  "_version": 49,
  "found": true,
  "jobId": "postgres-events"
}
```

Job API

The job API provides methods for managing running jobs and retrieving status about previous executions.

Permissions: To use the job API, you must ensure that the following *cluster-level* actions are granted by the security system:

- Abort: `cluster:admin/federate/connector/jobs/abort`
- Get: `cluster:admin/federate/connector/jobs/get`
- Running jobs: `cluster:admin/federate/connector/jobs/running/get`
- Job log: `cluster:admin/federate/connector/jobs/log/search`

Job management

The endpoint for job management is at `/_siren/connector/jobs`.

Running jobs statuses

The status of all running jobs can be retrieved by issuing a `GET` request as follows:

```
curl -XGET 'http://localhost:9200/_siren/connector/jobs/<type>'
```

The possible type value is:

- ingestion: This type is related to the ingestion jobs.

Running job status

The status of a job can be retrieved by issuing a `GET` request as follows:

```
curl -XGET 'http://localhost:9200/_siren/connector/jobs/<type>/<id>'
```

This API provides the status of the current running job if there is any, or the status of the last execution.

Body parameters:

- `status`: the status of the job.

Status parameters:

- `id`: the id of the job.
- `is_running`: a boolean value indicating if the job is running.
- `is_aborting`: an optional boolean value which indicates that the job is aborting.
- `start_time`: a timestamp with the starting time of the job.
- `end_time`: a timestamp with the ending time of the job.
- `infos`: textual information.
- `error`: an optional sequence of error messages.
- `state`: the current state of the job.
- `count`: the total number of processed records.
- `last_id`: the optional last known value of the primary key column.

Possible state values:

- `running`: the job is running.
- `aborting`: the job is aborting due to the user request.
- `aborted`: the job has been aborted.
- `error`: the job failed with an error.
- `successful`: the job was completed successfully.

JSON representation while a job is running:

```
{
  "_id": "postgres-events",
  "type": "ingestion",
  "found": true,
  "status": {
    "version": 1,
    "id": "postgres-events",
    "is_running": true,
    "start_time": 1538731228589,
    "infos": "The job is running.",
    "state": "running",
    "count": 3459,
    "last_id": "2289"
  }
}
```

JSON representation of a successfully completed job:

```
{
  "_id": "postgres-events",
  "type": "ingestion",
  "found": true,
  "status": {
    "version": 1,
    "id": "postgres-events",
    "is_running": false,
    "start_time": 1538733893554,
    "end_time": 1538733911829,
    "infos": "The job is done.",
    "state": "successful",
    "count": 10000,
    "last_id": "12219"
  }
}
```

JSON representation of a job who failed due to an error:

```
{
  "_id": "postgres-events",
  "type": "ingestion",
  "found": true,
  "status": {
    "version": 1,
    "id": "postgres-events",
    "is_running": false,
    "start_time": 1538730949766,
    "end_time": 1538730961293,
    "infos": "The job has failed.",
    "error": [
      "Could not execute datasource query [postgres].",
      "Failed to initialize pool: The connection attempt failed.",
      "The connection attempt failed.",
      "connect timed out"
    ],
    "state": "error",
    "count": 0
  }
}
```

Cancelling a running job

This API provides a method to cancel a running job.

```
curl -XPOST 'http://localhost:9200/_siren/connector/jobs/ingestion/<id>/_abort'
```

```
{
  "_id": "postgres-events",
  "type": "ingestion",
  "found": true,
  "status": {
    "version": 1,
    "id": "postgres-events",
    "is_running": false,
    "is_aborting": true,
    "start_time": 1538733800993,
    "end_time": 1538733805318,
    "infos": "The job has been aborted.",
    "state": "aborted",
    "count": 2220,
    "last_id": "2219"
  }
}
```


Searching on the job log

This API provides a method to retrieve the status of completed jobs. It is possible to pass parameters to filter the results.

```
curl -XGET 'http://localhost:9200/_siren/connector/jobs/_search'
```

Possible filter parameters:

- `start_time_from`: jobs which start time is greater than or equal to the passed value.
- `start_time_to`: jobs which start time is lower than or equal to the passed value.
- `type`: a filter on the job type.
- `state`: the state of the job status. See the job status description to get a list of possible values.
- `id`: the id of the job.

Request and result example:

```
curl -XGET 'http://localhost:9200/_siren/connector/jobs/_search?type=ingestion&id=pos
tgresevents&start_time_to=1539192173232'
```

```
{
  "hits": {
    "total": 1,
    "hits": [
      {
        "_id": "postgresevents11e247fa-ccb1-11e8-ad75-c293294ec513",
        "_source": {
          "ingestion": {
            "version": 1,
            "id": "postgresevents",
            "is_running": false,
            "start_time": 1539192150699,
            "end_time": 1539192151612,
            "infos": "The job is done.",
            "state": "successful",
            "count": 0
          }
        }
      }
    ]
  }
}
```


Sessions APIs

The Sessions APIs enable the management of user sessions. Siren Federate tracks the number of concurrent user sessions across the cluster. A user session must be specified for each search request with the header `X-Federate-Session-Id`. A same session id can be reused across multiple search requests.

Permissions: To use the sessions APIs, you must ensure that the following *cluster-level* actions are granted by the security system:

- Get: `cluster:admin/federate/sessions/get`
- Clear: `cluster:admin/federate/sessions/clear`

Get Sessions

The Get Sessions API allows to retrieve the list of the current active sessions.

```
curl -XGET 'http://localhost:9200/_siren/sessions'
```

The response includes the size of the session pool, the number of active sessions and the list of active session ids:

```
{
  "size": 5,
  "active": 2,
  "active_sessions_ids" : [ user_1, user_2 ]
}
```

Clear Sessions

Sessions are automatically removed when the session timeout since the last search request has been exceeded. However, it is recommended to clear the session as soon as the session is not being used anymore in order to free slots in the session pool:

```
curl -XDELETE 'http://localhost:9200/_siren/sessions/user_1'
```

or

```
curl -H 'Content-Type: application/json' -XDELETE
'http://localhost:9200/_siren/sessions' -d '
{
    "session_id" : "user_1"
}
'
```

Multiple session IDs can be passed as a comma separated list of values

```
curl -XDELETE 'http://localhost:9200/_siren/sessions/user_1,user_2'
```

or as an array:

```
curl -H 'Content-Type: application/json' -XDELETE
'http://localhost:9200/_siren/sessions' -d '
{
    "session_id" : [
      "user_1",
      "user_2"
    ]
}
'
```

License APIs

Federate includes a license manager service and a set of rest commands to register, verify and delete a Siren's license. By default, the Siren Community license is included.

Without a valid license, Federate will log a message to notify that the current license is invalid whenever a search request is executed.

> **Permissions:** To use this API, ensure that the *cluster-level* wildcard action `cluster:admin/federate/license/*` is granted by the security system.

Put License

The Put License API allows to upload a license to the cluster:

```
curl -XPUT 'http://localhost:9200/_siren/license'
```

Let's assume you have a Siren license named `license.sig`. You can upload and register this license in Elasticsearch using the command:

```
$ curl -XPUT -H 'Content-Type: application/json' -T license.sig
'http://localhost:9200/_siren/license'
---
acknowledged: true
```

Get License

The Get License API allows to retrieve and validate the license:

```
curl -XGET 'http://localhost:9200/_siren/license'
```

The response includes the content of the license as well as a summary of the license validation. If the validity check fails, a list of invalid parameters with a cause is provided:

```
{
  "license_content": {
    "description": "Siren Community License",
    "issue_date": "2019-01-29",
    "permissions": {
      "federate": {
        "max_concurrent_sessions": "1",
        "max_nodes": "1"
      },
      "investigate": {
        "max_dashboards": "12",
        "max_graph_nodes": "500",
        "max_virtual_indices": "5"
      }
    },
    "valid_date": "2020-01-29"
  },
  "license_validation": {
    "is_valid": false,
    "invalid_parameters": [
      {
        "parameter": "permissions.federate.max_nodes",
        "cause": "Too many nodes in the Federate cluster 2 > 1"
      },
      {
        "parameter": "permissions.federate.max_concurrent_sessions",
        "cause": "Too many concurrent user sessions in the Federate cluster 5 > 1"
      }
    ]
  }
}
```


Delete License

The Delete License API allows to delete a license from the cluster. Without license, the system will fall back to the Siren Community license.

```
curl -XDELETE 'http://localhost:9200/_siren/license'
```

Performance Considerations

Join types

Siren Federate offers three join strategies: the *hash join*, the *broadcast join* and the *index join*.

All strategies have advantages and disadvantages, but choosing the right one can help to optimize

system performance. For more information, see Configuring joins by type.

Numeric versus string attributes

Joining numeric attributes is more efficient than joining string attributes. If you are planning to join attributes of type `string`, we recommend to generate a murmur hash of the string value at indexing time into a new attribute, and use this new attribute for the join. Such index-time data transformation can be easily done using Logstash's `fingerprint` plugin.

Vectorized pipeline performance

Tuples collected will be transferred in one or more `packets`. The size of a packet has an impact on the performance. Smaller packets will take less memory but will increase cpu times on the receiver side since it will have to reconstruct a tuple collection from many small packets (especially for sorted tuple collection). By default, the size of a packets is set to 8MB, (which represents 1,048,576 tuples for a column of long datatype). The size can be configured using the setting key `siren.io.pipeline.max_packet_size` with a value representing the maximum size (in bytes) of a packet. For more information, see Vectorized Pipeline.

Using the `preference` parameter for search requests

To optimize cache utilization, Elasticsearch recommends using the `preference` parameter, which controls which shard copies on which to execute the search.

By default, Elasticsearch selects from the available shard copies in an unspecified order, taking the allocation awareness and adaptive replica selection configuration into account. However, it may sometimes be desirable to try and route certain searches to certain sets of shard copies. For example, the `preference` parameter could be set to a custom string value like a session or user id. This is very important in Siren Federate to better leverage the join query cache.

For more information, see Tune for search speed.

Caution when force-merging single-segment indices

The `search-project` task parallelizes its work by using a single worker per index segment. Therefore, caution must be exercised when considering a force-merge of an index.

Force-merging an index with a single segment impacts the `search-project` task's performance, as it will not be able to parallelize.

Troubleshooting guide

Installation error when extracting the plugin ZIP file

When you install the Siren Federate plugin , you might get the following error:

On Windows

```
elasticsearch-plugin install PATH-TO-SIREN-FEDERATE-PLUGIN\siren-federate-7.16.3-26.5
.zip
> Downloading PATH-TO-SIREN-FEDERATE-PLUGIN\siren-federate-7.16.3-26.5.zip
[=================================================] 100%
Exception in thread "main" java.nio.file.NoSuchFileException: PATH-TO-ELASTICSEARCH
\plugins.installing-3603227438462114792\plugin-descriptor.properties
at sun.nio.fs.WindowsException.translateToIOException(WindowsException.java:79)
at sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:97)
at sun.nio.fs.WindowsException.rethrowAsIOException(WindowsException.java:102)
at sun.nio.fs.WindowsFileSystemProvider.newByteChannel(WindowsFileSystemProvider.java
:230)
at java.nio.file.Files.newByteChannel(Files.java:361)
at java.nio.file.Files.newByteChannel(Files.java:407)
at java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.java:384)
at java.nio.file.Files.newInputStream(Files.java:152)
at org.elasticsearch.plugins.PluginInfo.readFromProperties(PluginInfo.java:162)
at org.elasticsearch.plugins.InstallPluginCommand.loadPluginInfo(InstallPluginCommand
.java:724)
at org.elasticsearch.plugins.InstallPluginCommand.installPlugin(InstallPluginCommand
.java:803)
at org.elasticsearch.plugins.InstallPluginCommand.install(InstallPluginCommand.java
:786)
at org.elasticsearch.plugins.InstallPluginCommand.execute(InstallPluginCommand.java
:232)
at org.elasticsearch.plugins.InstallPluginCommand.execute(InstallPluginCommand.java
:217)
at org.elasticsearch.cli.EnvironmentAwareCommand.execute(EnvironmentAwareCommand.java
:86)
at org.elasticsearch.cli.Command.mainWithoutErrorHandling(Command.java:124)
at org.elasticsearch.cli.MultiCommand.execute(MultiCommand.java:77)
at org.elasticsearch.cli.Command.mainWithoutErrorHandling(Command.java:124)
at org.elasticsearch.cli.Command.main(Command.java:90)
at org.elasticsearch.plugins.PluginCli.main(PluginCli.java:47)
```

On Linux

```
Exception in thread "main" java.nio.file.NoSuchFileException: PATH-TO-
ELASTICSEARCH/plugins/.installing-2425832270248497263/plugin-descriptor.properties
at sun.nio.fs.UnixException.translateToIOException(UnixException.java:86)
at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)
at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)
at sun.nio.fs.UnixFileSystemProvider.newByteChannel(UnixFileSystemProvider.java:214)
at java.nio.file.Files.newByteChannel(Files.java:361)
at java.nio.file.Files.newByteChannel(Files.java:407)
at java.nio.file.spi.FileSystemProvider.newInputStream(FileSystemProvider.java:384)
at java.nio.file.Files.newInputStream(Files.java:152)
at org.elasticsearch.plugins.PluginInfo.readFromProperties(PluginInfo.java:162)
at
org.elasticsearch.plugins.InstallPluginCommand.loadPluginInfo(InstallPluginCommand.jav
a:724)
at
org.elasticsearch.plugins.InstallPluginCommand.installPlugin(InstallPluginCommand.java
:803)
at
org.elasticsearch.plugins.InstallPluginCommand.install(InstallPluginCommand.java:786)
at
org.elasticsearch.plugins.InstallPluginCommand.execute(InstallPluginCommand.java:232)
at
org.elasticsearch.plugins.InstallPluginCommand.execute(InstallPluginCommand.java:217)
at
org.elasticsearch.cli.EnvironmentAwareCommand.execute(EnvironmentAwareCommand.java:86)
at org.elasticsearch.cli.Command.mainWithoutErrorHandling(Command.java:124)
at org.elasticsearch.cli.MultiCommand.execute(MultiCommand.java:77)
at org.elasticsearch.cli.Command.mainWithoutErrorHandling(Command.java:124)
at org.elasticsearch.cli.Command.main(Command.java:90)
at org.elasticsearch.plugins.PluginCli.main(PluginCli.java:47)
```

This error occurs because the command is being run on the distribution ZIP file, rather than on the plugin ZIP file inside it.

To resolve this error, complete the following steps:

1. Extract the Siren Federate distribution ZIP file into a local directory.
2. In the extracted directory, locate the plugin ZIP file named `siren-federate-7.16.3-26.5-proguard-plugin.zip`. The path to this plugin ZIP file is represented by `PATH-TO-SIREN-FEDERATE-PLUGIN` in the command that follows.
3. Run the installation command: `$ ./bin/elasticsearch-plugin install file:///PATH-TO-SIREN-FEDERATE-PLUGIN/siren-federate-7.16.3-26.5-proguard-plugin.zip`

Cannot start the buffer allocator service

The following error message is displayed:

```
elasticsearch.bootstrap.StartupException: BufferAllocatorException[Cannot start the
buffer allocator service];
```

This error occurs when the memory root limit is higher than the direct memory limit. The default direct memory limit is two-thirds of the maximum direct memory size of the JVM.

You can resolve this in one of the following ways:

- Open the `config/elasticsearch.yml` file and reduce the `siren.memory.root.limit` parameter to a value in bytes that is lower than the direct memory limit.
- Open the `jvm.options` file and increase the direct memory limit. For more information, see Setting off-heap memory.

Out of memory exception

The following error message is displayed:

```
out_of_memory_exception: Unable to allocate buffer of size.
```

This message indicates that off-heap memory allocation is used up. A join might require more memory than that which is available.

Open the `config/elasticsearch.yml` file and increase the `siren.memory.root.limit` parameter to a value in bytes for the root allocator.

Changing the thread pool queue size

You can modify the thread pools in Siren Federate.

To change the thread pool queue size, follow the instructions in the Elasticsearch documentation.

To verify how much thread pool queue is in use, follow the instructions in the Elasticsearch documentation.

Supported data types in a join

Siren Federate supports all primitive data types, however you must ensure that the data type of the joined fields across index patterns is the same.

For example, if you try to join a field from the pattern `index*`, but the field is an `integer` in `index1` while it is a `keyword` in `index2`, an error will result.

For more information, see Configuring joins by type.

Support for joining on the document ID

When supplying the `on` parameter to the join query, it is not possible to join on the `_id` meta field, as this field does not have `doc_values` enabled.

One solution is to index the `_id` in a secondary field with `doc_values` enabled, and use this field in the join.

You can duplicate the content of the `_id` field into another field that has `doc_values` enabled on the client side or use a set ingest processor.

For more information about how `doc_values` are scanned during a join, see Configuring joins by type.

Minimum memory requirements

If you are uncertain about the minimum memory that is required to use Siren Federate, it depends on the size of your data set and the size of the join.

For more information, see Configuring off-heap memory.

System performance

If the response time of search requests that involve joins is too long, try the following options to improve the performance of the join:

- Increase the number of nodes or shards in the index. For more information, see the Elasticsearch documentation.
- Configure a task timeout. For more information, see Search Request.

Release notes

7.16.3-26.5

Improvements

- Upgraded to Elasticsearch version 7.16.3.

7.16.2-26.4

Bug fixes

- Fixed an issue that prevented the execution of an msearch request containing joins on the same index with different algorithms.

7.16.2-26.3

Improvements

- Upgraded to Elasticsearch version 7.16.2. This Elasticsearch release includes fixes for CVE-2021-45105. For more information, please refer to ESA-2021-31.

7.16.1-26.2

Improvements

- Upgraded to Elasticsearch version 7.16.1. This Elasticsearch release includes fixes for CVE-2021-44228 and CVE-2021-45046. For more information about these vulnerabilities, refer to ESA-2021-31.

7.15.2-26.1

Improvements

- Upgraded to Elasticsearch version 7.15.2.

7.15.1-26.0

Improvements

- Upgraded to Elasticsearch version 7.15.1.
- Introduced vectorization of the project phase to increase throughput, including:
  - Refactoring of tuple processing into pipelines for all join algorithms and all data types.
  - Reducing the I/O overload by using a contiguous memory chunk.
  - Improving the vectorized tuples' column reader/writer performance.

Breaking changes

- Removed a deprecated property, `siren.io.netty.maxDirectMemory`, in favor of the `siren.memory.root.limit` setting.
- If `Search Guard` is the configured security system, the node-level setting `searchguard.allow_custom_headers: "siren.*"` must be added to each node of the cluster.
- Changed the format of the JSON response for `Cluster APIs Nodes Statistics` .

Bug fixes

- Fixed a join cache issue with Elastic X-Pack's templated queries when they include a placeholder such as `{{ _user: user.name }}`.
- Fixed a `ConcurrentModificationException` in the select phase following a recent change in

Elasticsearch version 7.15 involving the `FieldUsageTrackingDirectoryReader`.

- Adjusted the validation query in the datasource validation endpoint with the latest Avatica JDBC connector.
- Adjusted the `autoCommit` setting to work with the latest Avatica JDBC connector.
- Fixed a rare issue where a cluster is unable to initialize a Federate job due to a closed `RootBufferManager` on a node.
- Fixed a broken link to the `_sample` method in the connector APIs documentation.

Glossary

Many of the terms that are used in the Siren Federate documentation are also used in Elasticsearch. For more information, see the Elasticsearch glossary.

action

The type of request that can be executed on a cluster or an index. Actions are controlled and limited by user role permissions. For more information, see Configuring security for Siren Federate.]

API

The acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other.

broadcast join

A distributed join execution strategy, which copies the child set and duplicates it across every node of the cluster.

child set

During a join of indices A and B, a search is performed against index A as it is filtered by its relation to index B. In this example, the child set is index B (the filtering set) and the parent set is index A (the filtered set). **Note**: A set of documents can come from multiple indices.

cluster

One or more nodes that share the same cluster name.

datasource

An external source of data, such as a remote Elasticsearch cluster or a MySQL database behind an Avatica server. For more information, see Connecting to remote datasources.

document

A JSON document that is stored in Elasticsearch. A document is like a row in a table in a relational database. Each document is stored in an index and has a unique identifier associated with it.

Federate cluster

An Elasticsearch cluster that has the Siren Federate plugin installed.

federation

The process that maps different external database systems into a unified API so that it can be used for business intelligence (BI) or other analysis.

hash join

A distributed join execution strategy, where the two data sets are partitioned using a hash function across every node of the cluster, and where a hash table is used to find matching rows between the two inputs.

index

An optimized collection of JSON documents. An index is a logical namespace that maps to one or more primary shards and can have zero or more replica shards.

inner join

Enables the projection of arbitrary fields (including script fields and document's scores) from the child set, B, and combines them with the parent set, A. The projected fields and associated values of a document from set B are mapped to all of the documents from set A that satisfy the join condition. The result of the join is the parent set, A, augmented by the projected fields from the child set, B. See also, parent set, child set.

I/O

Disk I/O and caching occurs when the database engine reads and writes blocks containing records to and from a disk into memory. The next time the engine needs that block, it can access it from memory, rather than reading it from the disk.

join

A binary operator that is used to combine data from two sets of documents. The result of a join is the set of all combinations of documents in the two sets of documents that are equal on their common attribute names. For information about the different join strategies that are available, see Configuring joins by type.

join query

The type of query syntax to use when you want to perform a join. See also, query. For more information, see Query DSL .

left-side set

See parent set. Also known as the 'left index'.

node

An instance of Elasticsearch that belongs to a cluster. A node can combine different roles, such as a master-eligible node, a data node, an ingestion node, a transformation node, or a machine-learning (ML) node.

parallelization

A method of processing, whereby many operations are performed simultaneously - as opposed to serial processing, in which the computational steps are performed sequentially. Parallelization improves system performance through the simultaneous processing of various operations, such as loading data, building indexes, and evaluating queries.

parent set

During a join of indices A and B, a search is performed against index A as it is filtered by its relation to index B. In this example, the parent set is index A (the filtered set) and the child set is index B (the filtering set). Note: A set of documents may come from multiple indices.

partitioning

The process of breaking data in a database down into partitions. Each piece of data resides in exactly one partition. Partitioning is performed to ensure scalability, as entire data might not fit into a single node. Different partitions can reside on different nodes and each node can serve the queries with its own partition. See also, shard.

primary shard

Each document is stored in a single primary shard. When you index a document, it is indexed first on the primary shard, then on all replicas of the primary shard.

query

A request for information from Elasticsearch. A query represents a question, which is written in a way that Elasticsearch understands. A search consists of one or more queries combined.

reflection

The import and mapping of data to Elasticsearch from external datasources. A reflection is a recurrent and fully-managed ingestion that replicates the data from a datasource into an Elasticsearch index. See also, datasource.

replica shard

Each primary shard can have zero or more replicas. A replica is a copy of the primary shard, and has two purposes:

- Increased failover: A replica shard can be promoted to a primary shard if the existing primary shard fails.
- Improved performance: The `get` and `search` requests can be handled by primary or replica shards.

right-side set

See child set. Also known as the 'right index'.

semi-join

Filters the parent set (A), based on the child set (B). A semi-join returns the documents of A that satisfy the join condition with the documents of B. This is equivalent to the `EXISTS()` operator in SQL.

shard

A partition of an index in Elasticsearch. Each shard is held on a separate node to spread load. See also, partitioning and primary shard.

tuple

A single row that is composed of one or more columns, where one column is mapped to one field of a document. For example, a tuple can be a row that is composed of two elements, such as the

document identifier and the key value of the join condition. If a document has a multi-valued field, this will generate as many tuples as there are values.

virtual index

An Elasticsearch index that is created by the Federate plugin when mapping remote Elasticsearch clusters. The virtual index that is created does not contain the data itself. Instead, it contains information about the data source and its metadata. It is then used in `search` and `get` queries as any other index would be. For more information, see Connecting to remote datasources.