Siren Federate User Guide

Table of Contents

Introduction

The Siren Federate plugin extends Elasticsearch with (1) a federation layer to query external databases with the Elasticsearch API and (2) distributed join capabilities across indices and external databases.

Federation of External Databases

Siren Federate provides a module, called "Connector", which transparently maps external database systems to "Virtual Indices" in Elasticsearch. Requests to the Elasticsearch APIs, such as the `Mapping` or `Search` APIs, are intercepted by the Connector module. These requests are translated to the external database dialect and executed against the external database. This enables Siren Investigate to create and display dashboards for data located in external databases as if they were Elasticsearch's indices.

Distributed Joins Between Indices

Siren Federate extends the Elasticsearch DSL with a `join` query clause which enables a user to execute a join between indices (being virtual or not). The join capabilities are implemented on top of an in-memory distributed computing layer which scales with the number of nodes available in the cluster.

The current join capabilities is currently limited to a (left) semi-join between two set of documents based on a common attribute, where the result only contains the attributes of one of the joined set of documents. This join is used to filter one set of documents with a second document set. It is equivalent to the `EXISTS()` operator in SQL. Joins on both numerical and textual fields are supported, but the joined attributes must be of the same type. You can also freely combine and nest multiple joins using boolean operators (conjunction, disjunction, negation) to create complex query plans. It is fully integrated with the Elasticsearch API and is compatible with distributed environments.

How Does Siren Federate Join Compare With Parent-Child?

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 document and all of its children must live on the same shard, which limits its flexibility. The Siren Federate join removes this constraint and is therefore more flexible: it allows to join documents across shards and across indices.
- Thanks to the data locality of the Parent-Child model, joins are faster and more scalable. The Siren Federate join on the contrary needs to transfer data across the network to compute joins across shards, limiting its scalability and performance.

There is no "one size fits all" solution to this problem, and you need to understand your requirements to choose the proper 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. If on the contrary you need to query both directions of a data relationship, then the Siren Federate join might be more appropriate.

On Which Data Model It Operates

The most important requirement for executing a join is to have a common shared attribute between two indices. For example, let's take a simple relational data model composed of two tables, `Articles` and `Companies`, and of one junction table `ArticlesMentionCompanies` to encode the many-to-many relationships between them.

This model can be mapped to two Elasticsearch indices, `Articles` and `Companies`. 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 `Articles` table that refers to the primary key of the `Companies` table.

It should be straightforward for someone to write an SQL statement to flatten and map relationships into a single multi-valued field. We can see that, compared to a traditional database model where a junction table is necessary, the model is simplified by leveraging multi-valued fields.

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 is not inside the cluster, for example via JDBC connections.

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. We are leveraging the Apache Calcite engine to create a logical plan of the query, optimise the logical plan and execute a physical plan.
- A set of worker processes that 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.

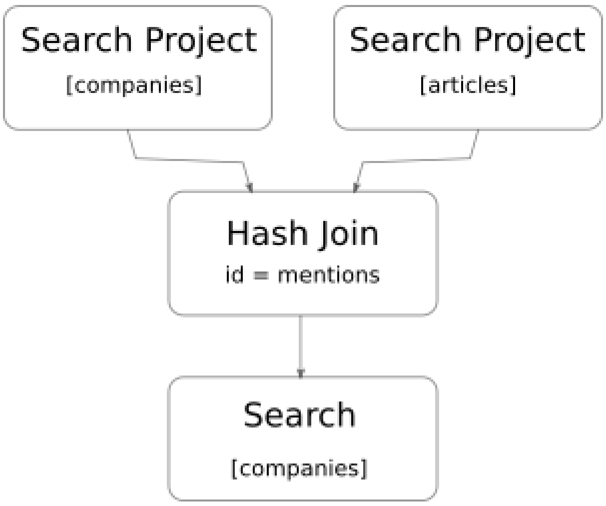
Distributed Join Workflow

When sending a (multi) search request with one or more nested joins, the node receiving the request will become the “Coordinator”. 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 represent a single type of operations, such as a `Search/Project` or `Join`, that is executed by a “Worker” on a node. A worker is a thread that will perform a task and report the outcome of the task to the coordinator.

For example, the following search request joining the index `companies` with `articles`:

```
GET /_siren/companies/search
{
   "query" : {
      "join" : {
        "type": "HASH_JOIN",
        "indices" : ["articles"],
        "on": ["id", "mentions"],
        "request" : {
          "query" : {
            "match_all": {}
          }
        }
      }
    }
}
```

will produce the following workflow:



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

This particular workflow enables Federate to push all the filtering predicates (e.g., terms, range, boolean queries) down to Elasticsearch, leveraging the indices for fast computation. The `Join` task is currently limited to compute the intersection of two different set of documents based on a join condition. This reduces the amount of data allocated in memory, the amount of data transferred across the network, and the workload performed by a task.

Query Planning & Optimisation

The coordinator node is leveraging 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 optimise the logical plan. We leverage both the Hep and Volcano engine to optimise 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. We are leveraging 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 collections of documents inter-connected by a common attribute, and execute a relational query across the two collections within the Elasticsearch environment.

Prerequisites

This guide requires that you have downloaded and installed the Elasticsearch 7.6.2 distribution on your computer. If you do not have an Elasticsearch distribution, you can run the following commands:

```
$ wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.6.2.zip
$ unzip elasticsearch-7.6.2.zip
$ cd elasticsearch-7.6.2
```

Installing the Siren Federate Plugin

Before starting Elasticsearch, you have to install the Siren Federate plugin. Assuming that you are in your Elasticsearch installation directory, you can run the following command:

```
$ ./bin/elasticsearch-plugin install file:///PATH-TO-SIREN-FEDERATE-PLUGIN/siren-
federate-7.6.2-20.0-plugin.zip
-> Downloading file:///PATH-TO-SIREN-FEDERATE-PLUGIN/siren-federate-7.6.2-20.0-
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
```

In case you want to remove the plugin, you can 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 collections of documents: Articles and Companies. An article is connected to a company with the attribute `mentions`. Articles will be loaded into the `articles` index and companies in the `companies` index. To load the dataset, run the following command:

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

$ curl -H 'Content-Type: application/json' -XPUT
'http://localhost:9200/_bulk?pretty&refresh=true' -d '
{ "index" : { "_index" : "articles", "_id" : "1" } }
{ "title" : "The NoSQL database glut", "mentions" : ["1", "2"] }
{ "index" : { "_index" : "articles", "_id" : "2" } }
```

```
{ "title" : "Graph Databases Seen Connecting the Dots", "mentions" : [] }
{ "index" : { "_index" : "articles", "_id" : "3" } }
{ "title" : "How to determine which NoSQL DBMS best fits your needs", "mentions" :
["2", "4"] }
{ "index" : { "_index" : "articles", "_id" : "4" } }
{ "title" : "MapR ships Apache Drill", "mentions" : ["4"] }

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


{
  "took" : 8,
  "errors" : false,
  "items" : [ {
    "index" : {
      "_index" : "articles",
      "_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/articles/_search?pretty' -d '{ ①
   "query" : {
      "join" : {                    ②
        "indices" : ["companies"],  ③
        "on" : ["mentions", "id"],  ④
        "request" : {               ⑤
          "query" : {
            "term" : {
              "name" : "orient"
            }
          }
        }
      }
    }
}'
```

① The target index (i.e. `articles`)

② The `join` query clause

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

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 `companies` (source set)

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

```
{
  "hits" : {
    "total" : 2,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "articles",
      "_id" : "1",
      "_score" : 1.0,
      "_source":{ "title" : "The NoSQL database glut", "mentions" : ["1", "2"] }
    }, {
      "_index" : "articles",
      "_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/companies/_search?pretty' -d '{
   "query" : {
      "join" : {
        "indices" : ["articles"],
        "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" : "companies",
      "_id" : "4",
      "_score" : 1.0,
      "_source":{ "id": "4", "name" : "MapR" }
    }, {
      "_index" : "companies",
      "_id" : "1",
      "_score" : 1.0,
      "_source":{ "id": "1", "name" : "Elastic" }
    }, {
      "_index" : "companies",
      "_id" : "2",
      "_score" : 1.0,
      "_source":{ "id": "2", "name" : "Orient Technologies" }
    } ]
  }
}
```

Set Up Federate

Configuring Logger

It is recommended to change the default Elasticsearch's log configuration `logger.action.level` from debug to warn in order to avoid spurious log messages whenever a search request is cancelled.

Off-heap memory management

Data is encoded in a columnar memory format and stored off-heap, reducing the pressure on the JVM and allowing fast and efficient analytical operations. Data reading is done directly from the off-heap storage and decoded on-the-fly using zero-serialization (removing any serialization overhead) and zero-copy memory (reducing 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 query, and per query operator, while having the inherent ability to terminate queries when the memory circuit breaker detects too many off-heap memory requests. 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 only used on the data nodes; master-only and coordinator nodes will not use off-heap memory.

Checking off-heap memory allocation

Federate provides a REST endpoint to retrieve statistics about the cluster (Nodes Statistics) which include off-heap memory allocation (Memory Information).

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 given that 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

The amount of off-heap memory available for the root allocator can be configured with the `siren.memory.root.limit` variable in `config/elasticsearch.yml`.

However, this value is limited by the **maximum direct memory limit** of Federate. If you want to set a larger limit for the root allocator, you must first manually set the maximum direct memory limit for Federate. This can be done using the `siren.io.netty.maxDirectMemory` variable.

For example, you can add this line to `config/jvm.options` of your Elasticsearch instance to increase the max direct memory limit to 2GB:

```
-Dsiren.io.netty.maxDirectMemory=2147483648
```

You can then configure the `siren.memory.root.limit` in the `config/elasticsearch.yml` to 2147483647 (i.e. 2147483648 - 1, as it is forbidden to use a limit that is greater than or equals to the max direct memory limit):

If you start the Elasticsearch instance, you can see the following info logs:

```
[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 on how Federate is configured, here we can see that:

① the max direct memory limit is correctly set to 2147483648

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 for accommodating the max direct memory limit for Federate, the JVM max heap memory limit, and the OS. If the sum of max direct memory limit for Federate and the JVM max heap memory limit do not leave enough memory for the OS, the OS might kill the Elasticsearch instance (*OOM killer* process on Linux systems).

64GB Machine

This is the recommended settings for cluster that needs to execute joins on a large amount of data

- 24 GB Heap (for Elasticsearch)
- 24 GB Off-heap (for Federate)
- 16 GB for the OS and OS Cache

config/jvm.options

```
-Xmx24g
-Dsiren.io.netty.maxDirectMemory=25769803776
```

config/elasticsearch.yml:

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

Otherwise, if the off-heap for 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 Federate)
- 16 GB for the OS and OS Cache

128GB Machine

- 32 GB Heap (for Elasticsearch)
- 64 GB Off-heap (for Federate)
- 32 GB for the OS and OS Cache

config/jvm.options

```
-Xmx32g
-Dsiren.io.netty.maxDirectMemory=68719476736
```

config/elasticsearch.yml:

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

If the off-heap for Federate is not fully used, it is then better to leave more memory for the OS and OS cache.

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 and statistical Volcano 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.enable`
: Enable or disable the Volcano statistical engine to select the most appropriate join algorithms. Defaults to `true`.

`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.

Memory

The memory module is responsible for allocating and managing chunks of off-heap memory. 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 (i.e., a Siren Federate search request) and is managing 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 and is managing 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 2/3 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 enables a simplified configuration for most common scenarios where only the root limit has to be configured. For more advanced scenarios, e.g., with multiple concurrent users, one might want to tune the job and task limits to avoid having a user executing a query that will consume all the available off-heap memory on the root level, leaving no memory for the queries executed by other users.

As a rule of thumb, one should never give more than half of the remaining OS memory to the siren root allocator, in order to leave some memory for the OS cache and 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.

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.

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 number of tuples in a packet. The packet size must be a power of 2. Defaults to 2^20 tuples.

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.

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 two new search actions, `/siren/<INDEX>/_search` that replaces the `/<INDEX>/_search` Elasticsearch action, and `/siren/<INDEX>/_msearch` that replaces the `/<INDEX>/_msearch` Elasticsearch action. Both actions are extensions of the original Elasticsearch actions and therefore support the same API. One must use these actions with the `join` query clause, as the `join` query clause is not supported by the original Elasticsearch actions.

> **Permissions:** the APIs listed in this section need to have the *cluster-level* wildcard action `cluster:internal/federate/*` granted by the security system, e.g., Search Guard.

Search API

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

Request

`GET /siren/<INDEX>/_search`

`POST /siren/<INDEX>/_search`

`GET /siren/_search`

`POST /siren/_search`

Path parameter

<index>

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

> **Permissions:** this API needs the *indices-level* wildcard action `indices:data/read/federate/search*` and `indices:data/read/federate/planner/search` to be granted by the security system, e.g., Search Guard.

Scroll API

The scroll API allows to paginate search hits. Similarly to Elasticsearch, you pass a `scroll` parameter to the Search API to set the duration of a scroll. Then to go through each pages or clear a scroll, you use the endpoint `/siren/_search/scroll/<SCROLL_ID>` instead of the `/_search/scroll/<SCROLL_ID>` indicated in the Elasticsearch documentation.

Permissions: in addition to the permissions for the Search API, this requires in addition the *indices-level* actions `indices:data/read/federate/scroll` and `indices:data/read/federate/scroll/clear` to be granted by the security system, e.g., Search Guard.

Multi Search API

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

Request

`GET /siren/<INDEX>/_msearch`

`POST /siren/<INDEX>/_msearch`

`GET /siren/_msearch`

`POST /siren/_msearch`

Path parameter

<index>

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

Permissions: this API needs the *indices-level* wildcard action `indices:data/read/federate/search*` and `indices:data/read/federate/planner/msearch` to be granted by the security system, e.g., Search Guard.

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`.

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:

  - if the `task_timeout` was set.
  - if a shard failed.

`query_plan` If the `debug` parameter was 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 simplifed query plan is displayed with information detailing the
causes of
the truncation.
```

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`.

> **Permissions:** this API needs the *cluster-level* action `cluster:admin/federate/job/cancel` to be granted by the security system, e.g., Search Guard.

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"
      }
    }
  ]
}
```

Query 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 the join algorithm to use. Valid values are either `BROADCAST_JOIN` or `HASH_JOIN`. If this parameter is not specified, the query planner will try to automatically select the optimal one.

`indices`
: The index names for the child collection. 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 collection, respectively. collection. Both fields must have the same datatype and must have the parameter `doc_values` set to true.

`request`
: The search request that is used to compute the set of documents of the child collection 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`.

```
GET /siren/target_index/_search
{
  "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`)

Scoring Capabilities

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

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`.

```
GET /siren/target_index/_search
{
  "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.

Project

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

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 collection 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, `companies` and `people`:

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

{ "index" : { "_index" : "people", "_type" : "person", "_id" : "1" } }
{ "id" : 1, "name" : "Alice", "age" : 31, "employed_by": 1 }
{ "index" : { "_index" : "people", "_type" : "person", "_id" : "2" } }
{ "id" : 2, "name" : "Bob", "age" : 42, "employed_by": 2 }
{ "index" : { "_index" : "people", "_type" : "person", "_id" : "3" } }
{ "id" : 3, "name" : "Carol", "age" : 26, "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/companies/_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" : "companies",
        "_type" : "company",
        "_id" : "2",
        "_score" : 0.0,
        "fields" : {
          "employees_age" : [
            42
          ]
        }
      },
      {
        "_index" : "companies",
        "_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 collection. It is composed of the following parameters:

`name`

The name of a field from a child collection 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 collection 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 boolean datatype, all numeric datatypes, and keyword datatype.

`script`
: The definition of a script supported by the Elasticsearch API. Projected fields from a child collection 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 collection 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 sort may 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/companies/_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" : "companies",
        "_type" : "company",
        "_id" : "1",
        "_score" : null,
        "_source" : {
          "id" : 1,
          "name" : "Acme"
        },
        "sort" : [
          57.0
        ]
      },
      {
        "_index" : "companies",
        "_type" : "company",
        "_id" : "2",
        "_score" : null,
        "_source" : {
          "id" : 2,
          "name" : "Bueno"
        },
        "sort" : [
          42.0
        ]
      }
    ]
  }
}
```

Scoring based on a projected field

A script-based scoring may 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 collection and aggregate them into the parent document as follows:

```
$ curl -H 'Content-Type: application/json'
'http://localhost:9200/siren/companies/_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" : "companies",
        "_type" : "company",
        "_id" : "1",
        "_score" : 2.0,
        "_source" : {
          "id" : 1,
          "name" : "Acme"
        }
      },
      {
        "_index" : "companies",
        "_type" : "company",
        "_id" : "2",
        "_score" : 1.0,
        "_source" : {
          "id" : 2,
          "name" : "Bueno"
        }
      }
    ]
  }
}
```

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.

```
GET /_siren/nodes/stats
GET /_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:** this API needs the *cluster-level* action `cluster:monitor/federate/nodes/stats` to be granted by the security system, e.g., Search Guard.

Memory Information

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

```
GET /_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": "Allocator(ROOT) 0/0/3750232064/17179869184
(res/actual/peak/limit)"
    }
  },
  "sKnVUBo9ShGzkl4GYih7BA": {
    "memory" : {
      "allocated_direct_memory_in_bytes" : 0,
      "allocated_root_memory_in_bytes": 0,
      "root_allocator_dump": "Allocator(ROOT) 0/0/0/17179869184
(res/actual/peak/limit)"
    }
  }
}
```

`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

`allocator_dump`
: Dump of the root allocator including the actual direct memory allocated, the peak and the limit.

Planner Information

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

```
GET /_siren/nodes/stats/planner
```

The response includes memory allocation statistics for each node 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

The `query_cache` flag can be set to retrieve information about the Siren’s query cache:

```
GET /_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
      }
    },
    "Dw06QS6oRbS3fEMazn511Q": {
      "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 snapshot of the query optimizer cache for a list of the cluster nodes.

```
GET /_siren/cache
GET /_siren/nodeId1,nodeId2/cache
GET /_siren/cache/clear
GET /_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:** this API needs the *cluster-level* action `cluster:monitor/federate/planner/optimizer/stats/get` to be granted by the security system, e.g., Search Guard.

Index APIs

The index APIs are used to manage individual indices.

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.

```
POST /<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:** the APIs listed in this section need to have the *cluster-level* wildcard action `cluster:internal/federate/*` granted by the security system, e.g., Search Guard.

Datasource API

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

We currently supports two types of datasources:

- **JDBC** to connect to any datasource providing a JDBC driver;
- **Elasticsearch** to connect to an Elasticsearch remote clusters.

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:** the creation of a datasource needs the *cluster-level* action `cluster:admin/federate/connector/datasource/put` granted by the security system, e.g., Search Guard.

JDBC datasource

```
PUT /_siren/connector/datasource/<id>
{
  "jdbc": {
    "username": "username",
    "password": "password",
    "driver": "com.db.Driver",
    "url": "jdbc:db://localhost:5432/default",
    "properties": {
      "ssl": true
    }
  }
}
```

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.

Elasticsearch datasource

```
PUT /_siren/connector/datasource/<id>
{
  "elastic": {
    "alias": "remotename"
  }
}
```

Elasticsearch configuration parameters:

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

Datasource retrieval

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

```
GET /_siren/connector/datasource/<id>
```

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

> **Permissions:** the retrieval of a datasource needs the *cluster-level* action `cluster:admin/federate/connector/datasource/get` granted by the security system, e.g., Search Guard.

Datasource deletion

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

```
DELETE /_siren/connector/datasource/<id>
```

> **Permissions:** the deletion of a datasource needs the *cluster-level* action `cluster:admin/federate/connector/datasource/delete` granted by the security system, e.g., Search Guard.

Datasource listing

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

```
GET /_siren/connector/datasource/_search
```

> **Permissions:** the listing of datasources needs the *cluster-level* action `cluster:admin/federate/connector/datasource/search` granted by the security system, e.g., Search Guard.

Datasource validation

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

```
POST /_siren/connector/datasource/<id>/_validate
```

> **Permissions:** the validation of a datasource needs the *cluster-level* action `cluster:admin/federate/connector/datasource/validate` granted by the security system, e.g., Search Guard.

Datasource catalog metadata

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

```
POST /_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 retrieve the metadata of a datasource, the *cluster-level* action `cluster:admin/federate/connector/datasource/metadata` should be granted by the security system, e.g., Search Guard.

Datasource field metadata

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

```
POST /_siren/connector/datasource/<id>/_resource_metadata?catalog=<catalog>&schema=<s
chema>&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 retrieve the field metadata of a datasource, the *cluster-level* action `cluster:admin/federate/connector/datasource/field-metadata` should be granted by the security system, e.g., Search Guard.

Datasource query sample

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

```
POST _siren/connector/datasource/<id>/_sample
{
  "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, the *cluster-level* action `cluster:admin/federate/connector/datasource/sample` should be granted by the security system, e.g., Search Guard.

Datasource transform suggestions

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

```
POST /_siren/connector/datasource/<id>/_transforms
{
    "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, the *cluster-level* action `cluster:admin/federate/connector/datasource/suggest/transform` should be granted by the security system, e.g., Search Guard.

Datasource type list

To get a list of supported connectors, issue a `GET` request as follows:

```
GET /_siren/connector/datasource
```


```
{
  "MySQL": {
    "driverClassName": "com.mysql.jdbc.Driver",
    "defaultURL":
"jdbc:mysql://{{host}}:{{port}}{{databasename}}?useLegacyDatetimeCode=false&useCursorF
etch=true",
    "defaultPort": 3306,
    "defaultQuery": "SELECT 1 AS N",
    "disclaimer": "This is a sample connection string, see the <a target=\"_blank\"
rel="noopener noreferrer" href=\"https://dev.mysql.com/doc/connector-
j/5.1/en/connector-j-reference.html\">MySQL Connector/J documentation</a> for further
information.",
    "virtualIndexSupported": true,
    "ingestionSupported": true
  },
  "PostgreSQL": {
    "driverClassName": "org.postgresql.Driver",
    "defaultURL": "jdbc:postgresql://{{host}}:{{port}}{{databasename}}",
    "defaultPort": 5432,
    "defaultQuery": "SELECT 1 AS N",
    "disclaimer": "This is a sample connection string, see the <a target=\"_blank\"
rel="noopener noreferrer" href=
\"https://jdbc.postgresql.org/documentation/94/connect.html\">PostgreSQL JDBC
documentation</a> for further information.",
    "virtualIndexSupported": true,
    "ingestionSupported": true
  }
}
```


Permissions: to get the type list, the *cluster-level* action `cluster:admin/federate/connector/datasource/types` should be granted by the security system, e.g., Search Guard.

Datasource driver list

To get a list of installed drivers, issue a `GET` request as follows:

```
GET /_siren/connector/datasource/_drivers
```

The result is a JSON document which contains the drivers list for each node part of the cluster.

The nodes are identified by their IDs. Use those IDs with the cluster API to get more information about the nodes.

The returned information for each driver is:

- `key`: The name of the JAVA class,
- `majorVersion`: The major version of the driver,
- `minorVersion`: The minor version of the driver,
- `jdbcCompliant`: Indicates if the driver is fully JDBC compliant.
- `infos`: The result of the "toString()" methods.

If any failure happened, the "failures" part of the JSON results contains the error message related to the failing node(s).


```
{
  "nodes": {
    "4PdaJD0BRd2lw47U6OAzog": {
      "org.apache.calcite.avatica.remote.Driver": {
        "majorVersion": 1,
        "minorVersion": 19,
        "jdbcCompliant": true,
        "infos": "org.apache.calcite.avatica.remote.Driver@6287f65f"
      },
      "org.apache.calcite.jdbc.Driver": {
        "majorVersion": 1,
        "minorVersion": 19,
        "jdbcCompliant": true,
        "infos": "org.apache.calcite.jdbc.Driver@1e592d50"
      }
    },
    "A4T54CAbT5qJgwzqtBvqIA": {
      "org.apache.calcite.avatica.remote.Driver": {
        "majorVersion": 1,
        "minorVersion": 19,
        "jdbcCompliant": true,
        "infos": "org.apache.calcite.avatica.remote.Driver@6287f65f"
      },
      "org.apache.calcite.jdbc.Driver": {
        "majorVersion": 1,
        "minorVersion": 19,
        "jdbcCompliant": true,
        "infos": "org.apache.calcite.jdbc.Driver@1e592d50"
      }
    },
    "OSYuK1ABQ0OC-XACY51ulA": {
      "org.apache.calcite.avatica.remote.Driver": {
        "majorVersion": 1,
        "minorVersion": 19,
        "jdbcCompliant": true,
        "infos": "org.apache.calcite.avatica.remote.Driver@6287f65f"
      },
      "org.apache.calcite.jdbc.Driver": {
```

```
        "majorVersion": 1,
        "minorVersion": 19,
        "jdbcCompliant": true,
        "infos": "org.apache.calcite.jdbc.Driver@1e592d50"
      }
    }
  },
  "failures": {
    "t5eC5E1WTDCrTai5I5Tk-g":
"[node_sc4][127.0.0.1:65429][cluster:admin/federate/connector/datasource/drivers/child
] disconnected"
  }
}
```

Permissions: to get the driver list, the *cluster-level* actions `cluster:admin/federate/connector/datasource/drivers` and `cluster:admin/federate/connector/datasource/drivers/child` should be granted by the security system, e.g., Search Guard.

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:

```
PUT /_siren/connector/index/<id>
{
  "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 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, the *indices-level* action `indices:admin/federate/connector/put` should be granted by the security system, e.g., Search Guard.

Virtual index deletion

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

```
DELETE /_siren/connector/index/<id>
```

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

> **Permissions:** to delete a virtual index, the *indices-level* action `indices:admin/federate/connector/delete` should be granted by the security system, e.g., Search Guard.

Virtual index listing

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

```
GET /_siren/connector/index/_search
```

> **Permissions:** to list virtual indices, the *indices-level* action `indices:admin/federate/connector/search` should be granted by the security system, e.g., Search Guard.

Ingestion API

Ingestion management

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

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.

```
PUT _siren/connector/ingestion/<id>
{
  "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:

```
GET /_siren/connector/ingestion/<id>
```

Ingestion deletion

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

```
DELETE /_siren/connector/ingestion/<id>
```

Ingestion listing

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

```
GET _siren/connector/ingestion/_all?status=[false|true]&detailed=[false|true]
```

NOTE `GET _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:

```
POST _siren/connector/ingestion/<id>/_validate
```

Run an ingestion job

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

```
POST _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.

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:

```
GET _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:

```
GET _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.

```
POST _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.

```
GET _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:

```
GET _siren/connector/jobs/_search?type=ingestion&id=postgresevents&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 enables the management of user sessions. Federate is tracking 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.

Get Sessions

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

```
GET /_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:

```
DELETE /_siren/sessions/user_1
```

or

```
DELETE /_siren/sessions
{
    "session_id" : "user_1"
}
```

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

```
DELETE /_siren/sessions/user_1,user_2
```

or as an array:

```
DELETE /_siren/sessions
{
    "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:** the *cluster-level* actions `cluster:admin/federate/license/*` need to be granted by the security system, e.g., Search Guard.

Put License

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

```
PUT /_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:

```
GET /_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.

```
DELETE /_siren/license
```

Connecting to Remote Datasources

The Siren Federate plugin provides the capability to query data in remote datasources through the Elasticsearch API by mapping tables to virtual indices.

The plugin stores its configuration in two Elasticsearch indices:

- `.siren-federate-datasources`: used to store the JDBC configuration parameters of remote datasources.

- `.siren-federate-indices`: used to store the configuration parameters of virtual indices.

Other indices are also used for different features: - `.siren-federate-ingestions`: used to store the ingestions configurations. - `.siren-federate-joblogs`: used to store logs of ingestion jobs.

Datasources and virtual indices can be managed using the REST API or the user interface available in Siren Investigate.

These indices are created automatically when required.

Connecting to JDBC datasources

Siren Federate provides the capability to query data from a remote JDBC databases.

Settings

In order to send queries to virtual indices the Elasticsearch cluster must contain at least one node enabled to issue queries over JDBC; it is advised to use a coordinating only node for this role, although this is not a requirement for testing purposes.

JDBC node settings

In order to enable JDBC on a node where the Siren Federate plugin is installed, add the following setting to `elasticsearch.yml`:

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

Then, create a directory named `jdbc-drivers` inside the configuration directory of the node (e.g. `elasticsearch/config` or `/etc/elasticsearch`).

Finally, copy the JDBC driver for your remote datasource and its dependencies to the `jdbc-drivers` directory created above and restart the node; see the JDBC driver installation and compatibility section for a list of compatible drivers and dependencies.

Common configuration settings

Encryption

JDBC passwords are encrypted by default using a predefined 128 bit AES key; before creating datasources, it is advised to generate a custom key by running the `keygen.sh` script included in the `siren-federate` plugin directory as follows:

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

The command will output a random base64 key; it is also possible to generate keys longer than 128 bit if your JVM supports it.

To use the custom key, the following parameters must be set in `elasticsearch.yml` on master nodes and on all the JDBC nodes:

- `siren.connector.encryption.enabled`: `true` by default, can be set to `false` to disable JDBC password encryption.
- `siren.connector.encryption.secret_key`: a base64 encoded AES key used to encrypt JDBC passwords.

Example `elasticsearch.yml` settings for a master node with a custom encryption key:

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

Example `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
```

Restart the nodes after changing the configuration to apply the settings.

JDBC driver installation and compatibility

The JDBC driver for your remote datasource and its dependencies must be copied to the `jdbc-drivers` subdirectory inside the configuration directory of JDBC nodes (e.g. `elasticsearch/config/jdbc-drivers`).

It is not required nor recommended to copy these drivers to nodes which are not enabled to execute queries.

You may create a sub-directory within `jdbc-drivers` to store a driver and to provide a custom security policy file for this driver. It is recommended to use this approach for drivers that come in multiple jars. A custom security policy file enables the definition of driver-specific permissions. The custom security policy file must be named `security.policy` and must be located inside the driver sub-directory. The following variables can be used within the policy file:

- `codebase.federate.common`: Path to the directory storing the `security.policy` (defaults to `jdbc-drivers` if the default `drivers-security.policy` file is used)
- `codebase.federate.${jar_name}`: Path to a driver JAR file. Here, `${jar_name}` refers to the name of the JAR file that is stored in the directory where the `security.policy` file is located. If the default `drivers-security.policy` file is used, then the system uses JAR files that are stored in the `jdbc-drivers` directory. For example:

```
grant codeBase "${codebase.federate.postgresql-42.2.5.jar}" {
  // Permissions for postgresql-42.2.5.jar
}
```

If a `security.policy` is placed in the main `jdbc-drivers` directory, then it overrides the default drivers-security.policy

Restart the JDBC node after copying the drivers.

Table 1. List of supported JDBC drivers

Name	JDBC class	Notes
PostgreSQL	org.postgresql.Driver	Download the latest JDBC 4.2 driver from https://jdbc.postgresql.org/download.html and copy the postgresql-<version>.jar file to the jdbc-drivers directory.
MySQL	com.mysql.jdbc.Driver	Download the latest GA release from https://dev.mysql.com/downloads/connector/j/, extract it, then copy mysql-connector-java-<version>.jar to the jdbc-drivers plugin directory. When writing the JDBC connection string, set the useLegacyDatetimeCode parameter to false to avoid issues when converting timestamps.
Oracle 12c+	oracle.jdbc.OracleDriver	Download the latest ojdbc8.jar from http://www.oracle.com/technetwork/database/features/jdbc/jdbc-ucp-122-3110062.html and copy it to the jdbc-drivers plugin directory.

Name	JDBC class	Notes
Spark SQL 2.2+	com.simba.spark.jdbc41.Driver	The Magnitude JDBC driver for Spark can be purchased at https://www.simba.com/product/spark-drivers-with-sql-connector/; once downloaded, extract the bundle, then extract the JDBC 4.1 archive and copy the following jars to the jdbc-drivers plugin directory: SparkJDBC41.jar commons-codec-<version>.jar hive_metastore.jar hive_service.jar libfb303-<version>.jar libthrift-<version>.jar ql.jar TCLIServiceClient.jar zookeeper-<version>.jar In addition, copy your license file to the jdbc-drivers plugin directory.
Neo4j	org.neo4j.jdbc.http.HttpDriver	Download the driver from https://mvnrepository.com/artifact/org.neo4j/neo4j-jdbc-driver/3.4.0 and copy the jar to the jdbc-drivers directory.

The following JDBC drivers are used by our customers but not officially supported. They are used as is.

Table 2. List of JDBC drivers used as is.

Name	JDBC class	Notes
Microsoft SQL Server 2014 or greater	com.microsoft.sqlserver.jdbc.SQLServerDriver	Download `sqljdbc_<version>_enu.tar.gz` from https://www.microsoft.com/en-us/download/details.aspx?id=55539, extract it, then copy `mssql-jdbc-<version>.jre8.jar` to the `jdbc-drivers` directory.
Sybase ASE 15.7+	com.sybase.jdbc4.jdbc.SybDriver OR net.sourceforge.jtds.jdbc.Driver	To use the FreeTDS driver, download the latest version from https://sourceforge.net/projects/jtds/files/, extract it, then copy `jtds-<version>.jar` to the `jdbc-drivers` directory. To use the jConnect driver, copy `jConnect-<version>.jar` from your ASE directory to the `jdbc-drivers` directory.
Presto	com.facebook.presto.jdbc.PrestoDriver	Download the latest JDBC driver from https://prestodb.io/docs/current/installation/jdbc.html and copy it to the `jdbc-drivers` plugin directory.
Dremio	com.dremio.jdbc.Driver	Download the jar at https://download.siren.io/dremio-jdbc-driver-1.4.4-201801230630490666-6d69d32.jar and copy it to the `jdbc-drivers` plugin directory.

Name	JDBC class	Notes

Impala	com.cloudera.impala.jdbc41.Driver	Download the latest JDBC bundle from https://www.cloudera.com/downloads/connectors/impala/jdbc/2-5-42.html, extract the bundle, then extract the JDBC 4.1 archive and copy the following jars to the `jdbc-drivers` plugin directory:
		`ImpalaJDBC41.jar`
		`commons-codec-<version>.jar`
		`hive_metastore.jar`
		`hive_service.jar`
		`libfb303-<version>.jar`
		`libthrift-<version>.jar`
		`ql.jar`
		`TCLIServiceClient.jar`
		`zookeeper-<version>.jar`

Operations on virtual indices

The plugin supports the following operations on virtual indices:

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

Search requests involving a mixture of virtual and normal Elasticsearch indices (e.g. when using a wildcard) are not supported and will be rejected; it is however possible to issue msearch requests containing requests on normal Elasticsearch indices and virtual indices.

When creating a virtual index, the plugin will create an empty 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.

When deleting a virtual index, the corresponding Elasticsearch index will not be removed.

Example of a search:

```
GET /siren/twitter/_search
{
  "query": {
    "match": {
      "user": "siren"
    }
  }
}
```

The API returns the following response:

```
{
  "took": 150,
  "timed_out": false,
  "hits": {
    "total" : {
        "value": 1,
        "relation": "eq"
    },
    "max_score": 1,
    "hits": [
      {
        "_index": "twitter",
        "_id": "0",
        "_score": 2,
        "_source": {
          "user": "siren",
          "date": "2019-11-16T12:12:12",
          "message": "trying out Siren",
          "likes": 0
        }
      }
    ]
  }
}
```

Type conversion

The plugin converts JDBC types to their closest Elasticsearch equivalent:

- String types are handled as `keyword` fields.
- Boolean types are handled as `boolean` fields.

- Date and timestamp are handled as `date` fields.
- Integer types are handled as `long` fields.
- Floating point types are handled as `double` fields.

Complex JDBC types which are not recognized by the plugin are skipped during query processing and resultset fetching.

Supported search queries

The plugin supports the following queries:

- match_all
- term
- terms
- range
- exists
- prefix
- wildcard
- ids
- bool

At this time the plugin provides no support for datasource specific full text search functions, so all these queries will work as if they were issued against `keyword` fields.

Supported aggregations

Currently the plugin provides support for the following aggregations:

Metric:

- Average
- Cardinality
- Max
- Min
- Sum

Bucket:

- Date histogram
- Histogram
- Date range
- Range
- Terms

- Filters

Only terms aggregations can be nested inside a parent bucket aggregation.

Troubleshooting

Cannot reconnect to datasource by hostname after DNS update

When the Java security manager is enabled, the JVM will cache name resolutions indefinitely; if the system you're connecting to uses round-robin DNS or 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 to `0` to avoid caching.

Connecting to Remote Elasticsearch Clusters

Siren Federate provides the capability to query data from an Elasticsearch remote cluster through the Remote Clusters Module and the Federate Connector APIs.

NOTE	This connector, unlike the JDBC connector, supports wildcard index patterns.

Configuring the Remote Cluster

To send queries from a cluster (let's call it the coordinator) to remote Elasticsearch clusters, the remote clusters must be configured as described in Configuring remote clusters.

The Siren Federate plugin has to be installed on the remote clusters.

This example shows how to set up the remote Elasticsearch clusters:

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

Configuring the Datasource

A datasource must first be defined as an alias to the remote cluster. Datasources are created in the coordinator cluster using the Federate Connector APIs.

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

Configuring the Virtual Index

Let’s assume our remote cluster `remotefederate` has indices called `logs-2019.01`, `logs-2019.02`, …, `logs-2019.12`.

Using a Wildcard Index Pattern

Let’s define a virtual index on the coordinator cluster that matches the wildcard index pattern `logs-*` using the Federate Virtual Index API:

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

Assuming the coordinator cluster has an index called `machines` which contains information on IP addresses on machines of interest, and that we would like to find out about the logs associated to these machines, you can execute the following Federate JOIN query to do so:

```
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

In order to take advantage of Federate with a remote cluster, at the moment a coordinator Federate cluster must run 7.6.2-20.0 up and the remote cluster must run Federate version from 7.6.2-20.0 up.

Search Guard Compatibility

The connector is compatible with Search Guard. One can define Search Guard users with roles to secure the remote clusters and the coordinator cluster.

Each cluster must have the same user that has permissions to access the cluster datasources, indices and virtual indices in order to properly execute Federate search requests on remote clusters.

Using curl and a Search Guard user called `admin`, the command would start like this:

```
curl -k -uadmin:password -X PUT https://localhost:9200/<some API request> ...
```

More information is available on the Search Guard website.

Known Limitations

Limitations for all the connectors

- Cross backend join currently supports only integer keys.
- Cross backend support has very different scalability according to the direction of the Join, a join which involves sending IDs to a remote system will be possibly hundreds of times less scalable (e.g. thousands vs millions) to one where the keys are fetched from a remote system.
- Currently cross cluster searches on virtual indices are not supported.

Limitations for the JDBC Connector

- Wildcards on virtual index names are not supported by any API; a wildcard search will silently ignore virtual indices.
- Comma-separated lists of index patterns which target virtual indices are not supported.
- Document-level security and field-level security are currently not supported.
- Only terms aggregations can be nested inside a parent bucket aggregation.
- The `missing` parameter in bucket aggregations is not supported.
- Scripted fields are not supported.
- When issuing queries containing string comparisons, the plugin does not force a specific collation, if a table behind a virtual indices uses a case insensitive collation, string comparisons will be case insensitive.
- Complex types are supported when their property types are scalar (text, numbers, boolean) or collections (list, map).
- Arrays of complex type are supported if the complex type meets the previous requirement.

Set Up Security

The Siren Federate plugin is compatible with Search Guard and Elastic X-Pack. You will find below instructions on how to configure both solutions for Federate.

Search Guard

We assume in this section that you are familiar with Search Guard, that Search Guard is installed in your cluster, and that you know how to configure users, roles and permissions. If not, please refer to the Search Guard documentation first.

Configuring Action Groups

Here is a list of action groups that are suitable for Federate.

sg_action_groups.yml

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

INDICES_ALL:
```

```
  allowed_actions:
  - "indices:*"

MANAGE:
  allowed_actions:
  - "indices:monitor/*"
  - "indices:admin/*"                                        ①

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

READ:                                                        ②
  allowed_actions:
  - "indices:data/read*"

VIEW_INDEX_METADATA:                                         ③
  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/*"                                      ④

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/scroll*"

CLUSTER_MANAGE:                                           ⑤
  allowed_actions:
  - CLUSTER_INTERNAL_FEDERATE
  - "cluster:admin/federate/*"
  - "indices:admin/aliases*"

CLUSTER_INTERNAL_FEDERATE:                                ⑥
  allowed_actions:
  - "cluster:internal/federate/*"
```

1. Federate's actions related to index administration are prefixed with `indices:admin/federate`
2. Federate's actions related to index read are prefixed with `indices:data/read/federate`
3. Grants permission to read index metadata, like getting field mapping
4. Federate's actions related to cluster monitoring are prefixed with `cluster:monitor/federate`
5. Federate's actions related to Federate administration are prefixed with `cluster:admin/federate`
6. All internal Federate's actions are prefixed with `cluster:internal/federate`

Configuring Role-Based Access Control

Given the action groups defined above, we can define two types of roles:

- the `federate_admin` role which can administrate Federate. For example, this role can manage license, virtual indices, ingestion jobs, etc.
- the `federate_user` role with read-only permissions which can execute Federate's search requests against one or more indices (virtual or not).

sg_roles.yml

```
federate_admin:
  cluster_permissions:
    - CLUSTER_MANAGE                    ①
    - CLUSTER_MONITOR                   ②
  index_permissions:
    - index_patterns:
      - 'logstash-*'
      allowed_actions:
        - MANAGE
        - READ
        - VIEW_INDEX_METADATA           ③

federate_user:
  cluster_permissions:
    - CLUSTER_INTERNAL_FEDERATE         ④
  index_permissions:
    - index_patterns:
      - 'companies'
      allowed_actions:
        - READ
        - VIEW_INDEX_METADATA           ③
```

1. Grants Federate cluster administration permissions.
2. Grants Federate cluster monitoring permissions.
3. Grants permissions to read index metadata. This is required given that the Federate's query engine will access index schema metadata using `indices:admin/mappings/fields/get` during the query evaluation.
4. Grants cluster-level permission for Federate's internal actions. This is required for every Federate users.

Securing Connector

When using Search Guard, Federate will need to authenticate as a user with all the permissions on the indices storing datasources and virtual indices configuration. The credentials of this user can be specified through the following node configuration settings:

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

Federate system role

If your cluster is protected by Search Guard, it is required to define a role with access to the Federate indices and internal operations and to create a Federate system user with this role.

Whenever a virtual index is created the Federate plugin creates a concrete Elasticsearch index with the same name as the virtual index: when starting up, the Federate plugin will check for missing concrete indices and will attempt to create them automatically.

sg_roles.yml

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

Then create a user with that role e.g., a user called `federate_system_user`.

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

=== .elasticsearch.yml

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

===

Example 1. JDBC node in a cluster with authentication and `federate_system_user` *user:*

elasticsearch.yml

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

Restart the nodes after setting the appropriate configuration parameters.

Administrative role

In order to manage, search, read datasources and virtual indices, it is required to grant the following cluster and indices-level permissions:

- `cluster:admin/federate/connector/*` which are given by the `CLUSTER_MANAGE` group;
- `indices:admin/federate/connector/*` which are included in the `MANAGE` group;
- `indices:admin/mappings/federate/connector/*` which are part of the `VIEW_INDEX_METADATA` group; and
- `indices:data/read/federate/connector/*` which are part of the `READ` group.

When a virtual index is defined, index-level write permissions are required because Federate creates a concrete index with the same name for interoperability with authentication plugins, unless such an index already exists.

For instance, if a MySQL datasource is defined and is named `db_mysql`, an index named `db_mysql` will be created. Then, the following `connector_admin` role can be created in order to manage/read/search it.

sg_roles.yml

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

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

Search role

In order to search virtual indices, a user needs `indices:data/read/federate/connector/*` permissions which are part of the `READ` group.

Keeping with the `db_mysql` virtual index example, a `connector_user` needs the following permissions granted:

sg_roles.yml

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

Elastic X-Pack Security

https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-security.html *federate system*

```
{
  "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*"
        ]
      }
    ]
  }
}
```

Admin role

```
{
  "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*"
        ]
      }
    ]
  }
}
```

Search role

```
{
  "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"
        ]
      }
    ]
  }
}
```

Performance Considerations

Join Types

Siren Federate includes different join strategies: “Broadcast Join” and “Hash Join”. Each one has its pros and cons and the optimal performance will depend on the scenario. By default, the Siren Federate planner will try to automatically pick the best strategy, but it might be best in certain scenarios to pick manually one of the strategies.

The Broadcast Join is best when filtering a large index with a small set of documents. The Hash Join is fully distributed and is designed to handle large joins. It scales horizontally (based on the number of nodes) and vertically (based on the number of cpu cores).

Siren Federate provides a fully distributed join algorithm: the Hash Join. The Hash Join is designed for leveraging multi-core architecture. This is achieved by creating many small data partitions during the Project phase. Each node of the cluster will receive a number of partitions that are dependent of the number of cpus. Partitions are independent from each other and can be processed independently by a different join worker thread. During the join phase, each worker thread will join tuples from one partition. The number of join worker threads scales automatically with the number of cpu cores available.

The Hash Join is performed in two phases: build and probe. The build phase creates a in-memory hash table of one of the relation in the partition. The probe phase then scans the second relation and probes the hash table to find the matching tuples.

Numeric vs 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.

Tuple Collector Settings

Tuple Collectors are sending batches of tuples of fixed size. The size of a batch has an impact on the performance. Smaller batches 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 batches (especially for sorted tuple collection). By default, the size of a batch of tuple is set to 1,048,576 tuples (which represents 8mb for a column of long datatype). The size can be configured using the setting key `siren.io.tuple.collector.batch_size` with a integer value representing the maximum number of tuples in a batch. :leveloffset: -1