SIREN

Siren Federate User Guide

Trademarks

Investigative Intelligence, Siren Alert, Siren Federate, Siren Investigate, and Siren Platform are trademarks of Sindice Ltd trading as Siren.

CentOS and Red Hat are trademarks of Red Hat Inc., registered in the U.S. and other countries.

CrunchBase is a trademark of CrunchBase Inc., registered in the U.S. and other countries.

Elasticsearch, Kibana, Logstash, and Packetbeat are trademarks of Elasticsearch BV, registered in the U.S. and other countries.

Excel, SQL Server, and Windows are trademarks of Microsoft Corporation, registered in the U.S. and other countries.

Java, JavaScript, and Oracle are trademarks of Oracle Corporation, registered in the U.S. and other countries.

Search Guard is a trademark of Floragunn GmbH, registered in the U.S. and in other countries.

All other trademarks are the property of their respective owners. All trademarks, registered trademarks and copyrighted terms in the Siren Investigate demonstration data set are the property of their respective owners.

Patents

Siren Federate patent pending:

- Irish Application Number S2018/0326

Table of Contents

Siren Federate

The Siren Federate plugin extends Elasticsearch with a federation layer to query external databases with the Elasticsearch API and 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 join capability 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 (https://www.elastic.co/guide/en/elasticsearch/guide/current/parent-child.html) 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 joining 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.

What data model does it operate on

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 using 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, optimize 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 serialization.

Distributed join workflow

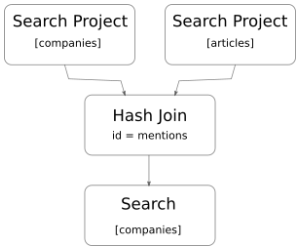
When sending a (multi) search request with one or more nested joins, the node receiving the request will become the "Coordinator". The coordinator node is in charge of controlling and executing a "Job" across the available nodes in the cluster. A job represents the full workflow of the execution of a (multi) search request. A job is composed of one or more "Tasks". A task represents 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:



Query 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 and optimization

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 optimize the logical plan. We leverage both the Hep and Volcano engine to optimize the logical plan using heuristic and statistical information. The logical plan is then transformed into a physical plan before being executed.

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

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

IO

The shuffling and transfer of data produced by a task is handled by a `Collector`. A collector will collect data, serialize it into a compact columnar data representation, and transfer it in the form of binary packets.

Different collector strategies are implemented that are adapted to different tasks. For example, in case of a `Hash Join`, a `Search/Project` task will use a collector with a hash partitioning strategy to create small data partitions and shuffle these partitions uniformly across the cluster.

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

The binary data packets are created, stored and manipulated off-heap. This helps to reduce unnecessary loads on the JVM and Garbage Collection when dealing with a large amount of data. 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

Unless you are using the complete Siren Platform package, you must **download** and installed the version of Elasticsearch that you want to use. If you do not have an Elasticsearch distribution, you can run the following commands where *<version>* is the version you want to use, for example 5.6.9:

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

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 where *<version>* is the Siren Federate version:

```
$ ./bin/elasticsearch-plugin install file:///<path to Siren Federate plugin>/siren-
federate-<version>-SNAPSHOT-plugin.zip
-> Downloading file:///<path to Siren Federate plugin>/siren-federate-<version>-
SNAPSHOT-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 data set for the purpose of this demo. The data set 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 data set, 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/article' -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/company' -d '
{
  "properties": {
    "id": {
      "type": "keyword"
    }
  }
}
'

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

{ "index" : { "_index" : "companies", "_type" : "company", "_id" : "1" } }
```

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

```
{
  "took" : 8,
  "errors" : false,
  "items" : [ {
    "index" : {
      "_index" : "articles",
      "_type" : "article",
      "_id" : "1",
      "_version" : 3,
      "status" : 200
    }
  },
  ...
}
```

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's filter, named `join`, that enables you to define such a query plan and a new search API `_search` that enables you to execute this query plan. Below is the command to run the relational query:

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

1. The `join` query clause.
2. The source indices (that is `companies`).

3. The clause specifying the paths for join keys in both source and target indices.
4. The search request that will be used to filter out companies.

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

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

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

```
$ curl -H 'Content-Type: application/json' 'http://localhost:9200/siren/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",
      "_type" : "company",
```

```
      "_id" : "4",
      "_score" : 1.0,
      "_source":{ "id": "4", "name" : "MapR" }
    }, {
      "_index" : "companies",
      "_type" : "company",
      "_id" : "1",
      "_score" : 1.0,
      "_source":{ "id": "1", "name" : "Elastic" }
    }, {
      "_index" : "companies",
      "_type" : "company",
      "_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 canceled.

JDBC drivers

JDBC driver jars for remote datasources and their dependencies (if any) must be copied to the plugin directory alongside other jars; once the jars have been copied, restart the node and ensure that the node starts correctly.

If the node fails to start because of a JAR Hell exception, remove the driver and its dependencies and restart the node.

Impala JDBC connector

The following jars should not be copied as they are already present in the default Elasticsearch path:

- `commons-logging`

Modules

Planner

The planner module is responsible in parsing a (multi) search request and generating a logical model. This logical model is then optimized 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.join`	Defines which distributed join algorithm to be selected when optimizing a request. Valid values are either `HASH_JOIN` or `MERGE_JOIN`, case-insensitive. Defaults to `HASH_JOIN`.
`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 1000000.

Memory

The memory module is responsible in 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 (that is, 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 750MB.
`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 limit 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.

Typically, you should never give more than half of the remaining OS memory to the siren root allocator, 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 in encoding, decoding and shuffling data across the nodes in the cluster.

Tuple collector

This module introduces the concept of `Tuple Collectors` which are responsible in 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 or Merge Join algorithm, 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.

Thread pools

Siren Federate introduces new thread pools:

`federate.planner`	For the query planner operations. Thread pool type is `scaling`.
`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_auto_queue_size` with a size of `int((# of available_processors * 3) / 2) + 1`, and initial queue_size of `1000`.

Connector

The Federate Connector module supports the following node configuration settings:

`siren.connector.datasources.index`	The index in which Federate will store datasource configurations.
`siren.connector.query.max_result_rows`	The maximum number of rows returned when executing a query on a remote datasource. Defaults to 10.

Search APIs

Siren Federate introduces two new search actions, `/siren/[INDICES]/_search` that replaces the `/[INDICES]/_search` Elasticsearch's action, and `/siren/[INDICES]/_msearch` that replaces the `/[INDICES]/_msearch` Elasticsearch's action. Both actions are extensions of the original Elasticsearch's actions and therefore supports 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.

Search API

The search API allows you to execute a search query and get back search hits that match the query. The endpoint for it is `/siren/[INDICES]/_search`.

Multi search API

The multi search API allows you to execute several search requests within the same API. The endpoint for it is `/siren/[INDICES]/_msearch`.

Search request

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

Parameters

In addition to the parameters supported by Elasticsearch's search API, Federate's 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. If the `task_timeout` was activated, it will include the flag `is_pruned` to indicate that the search results are pruned and probably incomplete. If the `debug` parameter was enabled, it will also include the information and statistics about the query plan execution.

Cancelling a Request

A search or a multi search request can be canceled explicitly by a user through HTTP headers. In order to do so, you need to pass an `X-Opaque-Id` header which is used to identify the request. The endpoint for canceling a request is `/_siren/job/<ID>/_cancel`.

Usage

Let's identity a search request with the ID `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 canceled 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`, `HASH_JOIN` or `MERGE_JOIN`. If this parameter is not specified, the query planner will try to automatically select the optimal one.
`indices`	The index names that will be joined with the source indices. Defaults to all indices.
`indices`	The index types that will be joined with the source indices. Defaults to all types.
`on`	An array specifying the field paths for join keys in both source and target indices. Both fields must have the same datatype with the parameter `doc_values` set to true. You should not join fields based on the `text` datatype.
`request`	The search request that will be used to compute the set of documents on the source before performing the join.

Example

In this example, we will join all the documents from `index1` with the documents of `index2` using the `HASH_JOIN` algorithm. The query first filters documents from `index2` 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 `index1`.

```
GET /siren/index1/_search
{
  "join" : {
    "type": "HASH_JOIN",
    "indices" : ["index2"],
    "types" : ["type"],
    "on" : ["foreign_key", "id"],
    "request" : {
      "query" : {
        "terms" : {
          "tag" : [ "aaa" ]
        }
      }
    }
  }
}
```

Scoring capabilities

The `join` filter has not scoring support and will return a constant score.

Compatibility with nested query

The `join` filter within a `nested` query is currently 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/index1/_search
{
  "nested" : {
    "path" : "nested_obj",
    "query" : {
      "join" : {
        "indices" : ["index2"],
        "types" : ["type"],
        "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.

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.

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 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)"
    }
  },
  "sKnVUBo9ShGzk14GYih7BA": {
    "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 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
        }
      }
    }
  }
}
```

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.

Connecting to JDBC 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.

You should restrict access to these indices to the Federate user.

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.

Settings

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

JDBC node settings

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 folder named `jdbc-drivers` inside the configuration folder of the node (for example `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 switch off 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.

Cluster wide settings

The following parameters can be set in `elasticsearch.yml` on JDBC nodes or by using the Elasticsearch cluster update settings API:

- `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. Set to `false` to switch off the use of unions in nested aggregations.
- `siren.connector.query.max_result_rows`: The maximum number of rows that will be retrieved from a resultset when performing a join across datasources. Defaults to `50000`.
- `siren.connector.query.max_bucket_queries`: The maximum number of JDBC queries that will be generated to compute aggregation buckets. Defaults to `500`.

Additional node settings

The following settings can be used to tune query processing on JDBC enabled nodes:

- `siren.connector.pool.size`: the number of threads that will be allocated to process the execution of queries to remote datasources; by default it is set to `int((number of available_processors * 3) / 2) + 1`.
- `siren.connector.pool.queue`: the maximum number of requests that should be queued if all the threads are busy. Defaults to `40`.

Authentication

The Federate server role

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

For interoperability with these plugins, 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.

Sample Search Guard role definition:

```
federateserver:
  cluster:
    - "indices:admin/aliases"
  indices:
    ?siren-federate-datasources:
      '*':
        - ALL
    ?siren-federate-indices:
      '*':
        - ALL
    ?siren-federate-target:
      '*':
        - ALL
```

Sample X-Pack role definition:

```
{
  "cluster": [
    "monitor",
    "cluster:admin/siren/connector"
  ],
  "indices" : [
    {
      "names" : [ "*" ],
      "privileges" : [ "create_index", "indices:data/read/get", "indices:admin/siren/
connector" ]
    },
    {
      "names" : [ ".siren-federate-*" ],
      "privileges" : [ "all", "indices:admin/siren/connector" ]
    }
  ]
}
```

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

Example `elasticsearch.yml` settings for a master node in a cluster with authentication and `federateserver` user:

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

Example `elasticsearch.yml` settings for a JDBC node in a cluster with authentication and `federateserver` user:

```
siren.connector.username: federateserver
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

To manage datasources and virtual indices, it is required to grant the `cluster:admin/siren/connector/*` permissions at the cluster level.

In addition, the user must have the `indices:admin/siren/connector/*` and `indices:data/siren/connector/*` permissions on all the index names that he's allowed to define, in addition to create, write, read and search permissions.

Write permissions are required because when a virtual index is defined the plugin will create a concrete Elasticsearch index with the same name for interoperability with authentication plugins, unless such an index already exists.

Example Search Guard role allowed to manage virtual indices starting with `db-`:

```
sirenadmin:
  cluster:
    - SIREN_CLUSTER
    - cluster:admin/plugin/siren/license/put
    - cluster:admin/plugin/siren/license/get
    - cluster:admin/siren/connector/&amp;#42;
  indices:
    'db-*':
      '*':
        - SIREN_READWRITE
        - indices:admin/create
        - indices:admin/siren/connector/*;
    '*':
      '*':
        - SIREN_COMPOSITE
```

Example X-Pack role allowed to manage virtual indices starting with `db-`:

```
{
  "cluster": [
    "cluster:admin/siren/connector"
    "cluster:admin/plugin/siren/license",
    "cluster:siren/internal",
    "manage"
  ],
  "indices" : [
    {
      "names" : [ "*" ],
      "privileges" : [ "indices:siren/mplan" ]
    },
    {
      "names" : [ "db-*" ],
      "privileges" : [
        "read",
        "create_index",
```

```
        "view_index_metadata",
        "indices:data/siren",
        "indices:siren",
        "indices:admin/version/get",
        "indices:admin/get",
        "indices:admin/siren/connector"
      ]
    }
  ]
}
```

Search role

In order to search virtual indices, users must have the `indices:data/siren/connector/*` permissions on these indices in addition to standard read and search permissions.

Example Search Guard role allowed to search virtual indices starting with `db-`:

```
sirenuser:
  cluster:
    - SIREN_CLUSTER
  indices:
    '*':
      '*':
        SIREN_COMPOSITE
    'db-*':
      '*':
        - SIREN_READONLY
        - indices:data/siren/connector*;
```

Example X-Pack role allowed to search virtual indices starting with `db-`:

```
{
  "cluster": [
    "cluster:admin/plugin/siren/license/get",
    "cluster:siren/internal"
  ],
  "indices" : [
    {
      "names" : [ "*" ],
      "privileges" : [ "indices:siren/mplan" ]
    },
    {
      "names" : [ "db-*" ],
      "privileges" : [
        "read",
        "view_index_metadata",
        "indices:data/siren",
        "indices:siren",
        "indices:admin/version/get",
        "indices:admin/get"
      ]
    }
  ]
}
```

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 neither required nor recommended to copy these drivers to nodes which are not enabled to execute queries.

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.
Microsoft SQL Server 2014 or greater	com.microsoft.sqlserver.jdbc.SQLServerDriver	Download `sqljdbc_<version>_enu.tar.gz` from https://docs.microsoft.com/en-us/sql/connect/jdbc/download-microsoft-jdbc-driver-for-sql-server?view=sql-server-2017#available-downloads-of-jdbc-driver-for-sql-server, 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.
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.
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.

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.

Dremio	com.dremio.jdbc.Driver	Download the jar at https://download.siren.io/dremio-jdbc-driver-1.4.4-20180123063049066-6d69d32.jar and copy it to the `jdbc-drivers` plugin directory.

Impala	com.cloudera.impala.jdbc41.Driver	Download the latest JDBC bundle from https://www.cloudera.com/downloads/connectors/impala/jdbc/2-6-4.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`

Restart the JDBC node after copying the drivers.

API

Datasource management

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

Datasource creation and modification

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

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

Body parameters:

- `jdbc`: The JDBC configuration of the datasource.

JDBC configuration parameters:

- `driver`: The class name of the JDBC driver.
- `url`: the JDBC URL of the datasource.
- `username` (optional): The username that will be passed to the JDBC driver when getting a connection.
- `password` (optional): The password that will be passed to the JDBC driver when getting a connection.
- `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.

Datasource deletion

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

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

Datasource listing

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

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

Datasource validation

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

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

Virtual index management

Virtual index creation and modification

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

```
PUT /_siren/connector/index/<id>
{
  "datasource": "ds",
  "catalog": "catalog",
  "schema": "schema",
  "resource": "table",
  "key": "id"
}
```

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.

Virtual index deletion

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

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

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

Virtual index listing

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

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

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 (for example, 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 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.

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.

Known Limitations

- 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 (for example, thousands compared to millions) to one where the keys are fetched from a remote system.
- 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.
- Wildcards on virtual index names are not supported by any API; a wildcard search will silently ignore virtual indices.
- Currently cross cluster searches on virtual indices are not supported.

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.

License API

Federate includes a license manager service and a set of rest commands to register, verify and delete a Siren's license.

Without a valid license, Federate will log a message to notify that the current license is invalid at every request.

Usage

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
```

You can then check the status of the license using the command:

```
$ curl -H 'Content-Type: application/json' 'http://localhost:9200/_siren/license'
{
  "license" : {
    "content" : {
      "valid-date" : "2016-05-16",
      "issue-date" : "2016-04-15",
      "max-nodes" : "12"
    },
    "isValid" : true
  }
}
```

To delete a license from Elasticsearch, you can use the command:

```
$ curl -XDELETE 'http://localhost:9200/_siren/license'
{"acknowledged":true}
```

Set up Security

Connector

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

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

Search Guard

The following snippet can be added to `sg_roles.yml` to define a `federateserver` role with all the required permissions on connector indices.

```
federateserver:
  indices:
    ?siren-federate-datasources:
      '*':
        - ALL
```

The following snippet can be added to `sg_roles.yml` to define a `federateuser` role with all the required permissions to manage datasources:

```
federateuser:
  cluster:
    - "cluster:admin/siren/connector/datasource/*"
```

Performance considerations

Join types

Siren Federate includes different join strategies: “Broadcast Join”, “Hash Join” and “Merge 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 and Merge Join are fully distributed and are designed to handle large joins. They both scales horizontally (based on the number of nodes) and vertically (based on the number of CPU cores). Currently, the Hash Join usually performs better in many scenarios compared to the Merge Join.

Siren Federate provides two fully distributed join algorithms: the Hash Join and the Sort-Merge Join. Each one 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 CPU cores. Partitions are independent of 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 an 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.

The Sort-Merge Join instead requires a sort phase of the two relations during the project phase. It then performs a linear scan over the two sorted relations to find the matching tuples.

Compared to the Hash Join, the Sort-Merge Join does not require additional memory since it does not have to build an in-memory hash table. However, it requires a sort operation to be executed during the project phase. It is in fact trading CPU for memory.

Numeric or string attributes

Joining numeric attributes is more efficient than joining string attributes. If you are planning to join attributes of type `string`, you should 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 by using Logstash’s `fingerprint` plugin (https://www.elastic.co/guide/en/logstash/current/plugins-filters-fingerprint.html).

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 1048576 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 an integer value representing the maximum number of tuples in a batch.

Glossary

acyclic	A graph without a cycle.
allocator	A process that allocates resources.
application programming interface (API)	A set of definitions that enable one piece of software to communicate with another.
authenticator	A process that performs authentication.
backend	Software running on a server that is not directly accessed by a user.
bitwise	An operation that modifies the individual bits of binary numeral.
choropleth	A map with areas shaded in proportion to a statistical variable. Also known as a *regionmap*.
comention	A pair of data items occurring together.
datasource	A connection to a database.
decrypt	To convert encrypted data to plain text.
deserialization	Creating an object from structured data.
formatter	A process that performs formatting.
geohash	An alphanumeric string that encodes a geographic location.
geopoint	A geographic location, typically expressed in latitude and longitude.
heatmap	A graphical data display using colors to represent individual values.
heteroscedastic	The property of a variable whose variability is unequal across the range of values predicted by a second variable.
hostname	A domain name that can be translated into an IP address.
iframe	A frame used to place one HTML document inside another.
jitter	Deviation from standard distribution.
keystore	A repository of security certificates. See *truststore*.
partitioner	A process that performs partitioning.

picosecond	One thousandth of a nanosecond.
regionmap	A map with areas shaded in proportion to a statistical variable. Also known as a *choropleth* map.
rollup	An index of preselected fields.
templated	The property of having a preset format.
templating	Applying a template. See *templated*.
tilemap	A geographic map overlaid with circles keyed to data specified in buckets.
truststore	A repository of certificates issued by Certificate Authorities. See *keystore*.
truthy	A value that returns `true` when treated as a Boolean value.
unescaped	Used to describe raw values that have not been encoded to avoid ambiguity.
validator	A process that performs validation.