Sensei

A Distributed Elastic Realtime Searchable Database

Sensei Development Team

Sensei: A Distributed Elastic Realtime Searchable Database

Sensei Development Team

$Id: sensei.xml,v 1.2 2011/04/25 08:56:30 bcui Exp $

Table of Contents

List of Figures

Chapter 1. Introduction

Sensei is a distributed database that is designed to handle the following type of query:

```
SELECT f1,f2...fn FROM members
WHERE c1 AND c2 AND c3..
MATCH (fulltext query, e.g. "java engineer")
GROUP BY fx,fy,fz...
ORDER BY fa,fb...
LIMIT offset,count
```

Design Considerations

- **Data**
  - Fault tolerance - when one replication is down, data is still accessible
  - Durability - N copies of data is stored
  - Through-put - Parallelizable request-handling on different nodes/data replicas, designed to handle internet traffic
  - Consistency - Eventally consistent
  - Data recovery - each shared/replica is noted with a watermark for data recovery
  - Large dataset - designed to handle 100s millions - billions of rows

- **Horizontally Scalable**
  - Data is partitioned - so work-load is also distributed
  - Elasticity - Nodes can be added to accomodate data growth
  - Online expansion - Cluster can grow while handling online requests
  - Online cluster management - Cluster topology can change while handling online requests
  - Low operational/maintenance costs - Push it, leave it and forget it.

- **Performance**
  - low indexing latency - real-time update
  - low search latency - millisecond query response time
  - low volatility - low variance in both indexing and search latency

- **Customizability**
  - plug-in framework - custom query handling logic
  - routing factory - custom routing logic, default: round-robin
  - index sharding strategy - different sharding strategy for different applications, e.g. time, mod etc.

Comparing to Traditional RDBMS

RDBMS:

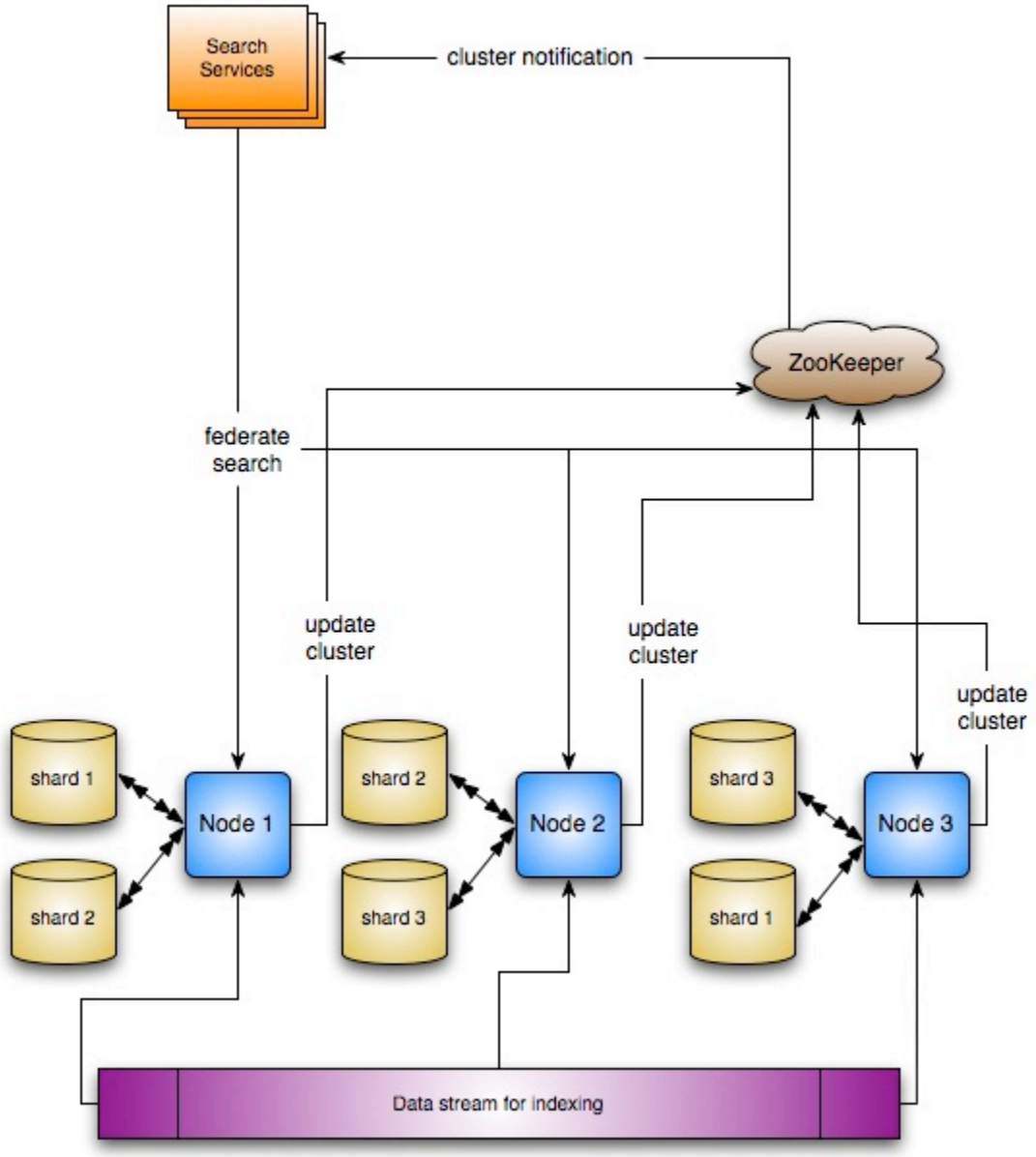
- vertically scaled
- strong ACID guarantee
- relational support
- performance cost with full-text integration
- high query latency with large dataset, esp. Group By
- indexes needs to be built for all sort possibilities offline

Sensei:

- horizontally scaled

- relaxed Consistency with high durability guarantees
- data is streamed in, so Atomicity and Isolation is to be handled by the data producer
- full-text support
- low query latency with arbitrarily large dataset
- dynamic sorting, index is already built for all sortable fields and their combinations

Architectural Diagram



Chapter 2. Getting Started

Overview

- Realtime indexing/searching
- Cluster management
- Automatic and configuration data partitioning
- Support for structured and faceted search

Prerequisites

- Java 1.6 or higher
- maven 2.2.1 or higher
- zookeeper 3.2.0 (http://hadoop.apache.org/zookeeper/)

Embedded Technologies:

- bobo-browse http://sna-projects.com/bobo/
- zoie http://sna-projects.com/zoie/
- lucene http://lucene.apache.org/
- norbert http://sna-projects.com/norbert/
- spring http://www.springsource.com/

Details

Build

1. Check out trunk:

```
git clone git://github.com/javasoze/sensei.git sensei-trunk
```

2. Simply do:

```
ant
```

ZooKeeper

In order to run the sample of sensei-search, you have to run an instance of **ZooKeeper** first.

You may download it from http://hadoop.apache.org/zookeeper/.

Using the sample configuration file in `zookeeper-3.2.0/conf` by copying `zookeeper-3.2.0/conf/zoo_sample.cfg` to `zookeeper-3.2.0/conf/zoo.cfg` and start an instance of zookeeper by running

```
zookeeper-3.2.0/bin/zkServer.sh start
```

For details, see http://hadoop.apache.org/zookeeper/docs/current/zookeeperStarted.html

Starting Sensei Nodes

Do `bin/start-sensei-node.sh` to start a server node. This command takes 1 argument: `conf.dir`, which contains all configuration information for a given Sensei node.

An example command-line that will work to fire up a single sensei node with some sample data:

```
bin/start-sensei-node.sh conf
```

Note

Do not expect to see any logs after running this command. If you run it, have zookeeper up and running, a REST server (as discussed below) will also be started, and you will be able to get some sample search results.

Web Application and Restful end-point:

A restful end-point along with a web interactive client would be started as well:

- restful endpoint: http://localhost:8080/sensei?q=
- web client: http://localhost:8080

For details, please checkout: Restful JSON End-point & Client Application

Starting Client

After you start at least one node, you can run `bin/sensei-client.sh client-conf` to start a client. (Edit `client-conf/sensei-client.conf` to change properties)

Type help to see command list:

```
help - prints this message
exit - quits
nodes - prints a list of node information
query <query string> - sets query text
facetspec <name>:<minHitCount>:<maxCount>:<sort> - add facet spec
page <offset>:<count> - set paging parameters
select <name>:<value1>,<value2>... - add selection, with ! in front of value ind
sort <name>:<dir>,... - set sort specs
showReq: shows current request
clear: clears current request
clearSelections: clears all selections
clearSelection <name>: clear selection specified
clearFacetSpecs: clears all facet specs
clearFacetSpec <name>: clears specified facetspec
browse - executes a search
```

Chapter 3. Sensei Demo

Overview

We feel the best way to learn a new system is through examples.

Sensei comes with a sample application and this page aims to provide an anatomy of the Sensei car demo and to help new-comers in building a Sensei application.

File layout:

- configuration files: `conf/*`
- data file: `data/cars.json`
- output index: `index/`
- web-app: `src/main/webapp/`

Run The Demo

Build:

```
ant
```

Make sure zookeeper is running:

```
$ZK_home/bin/zkServer.sh start
```

Run:

```
./bin/start-sensei-node conf/
```

URLs

- Sensei Web Client: http://localhost:8080/
- Demo: http://localhost:8080/
- Rest Endpoint: http://localhost:8080/sensei

Screenshots

Figure 3.1. Sensei Web Client

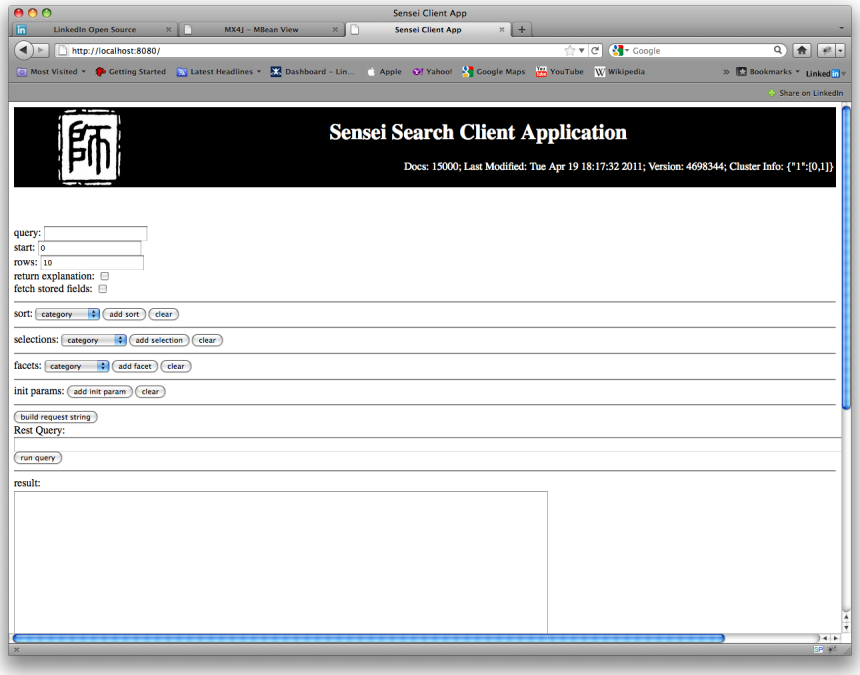


Figure 3.2. Demo

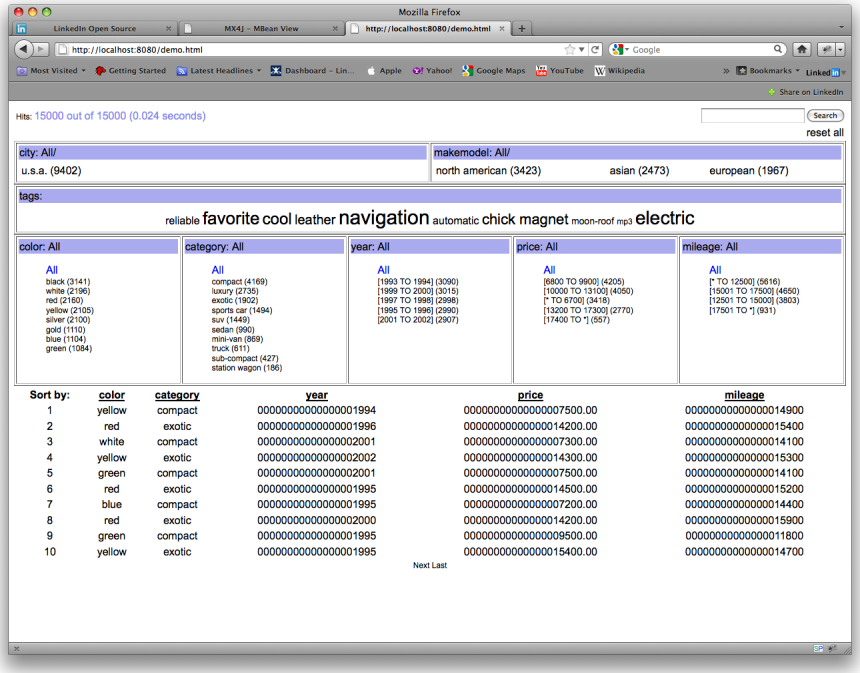
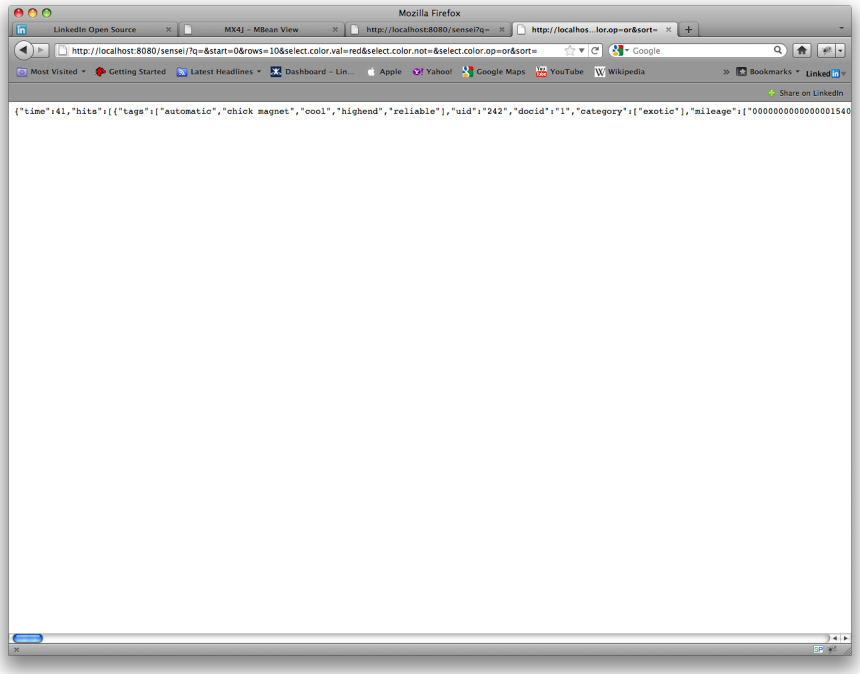
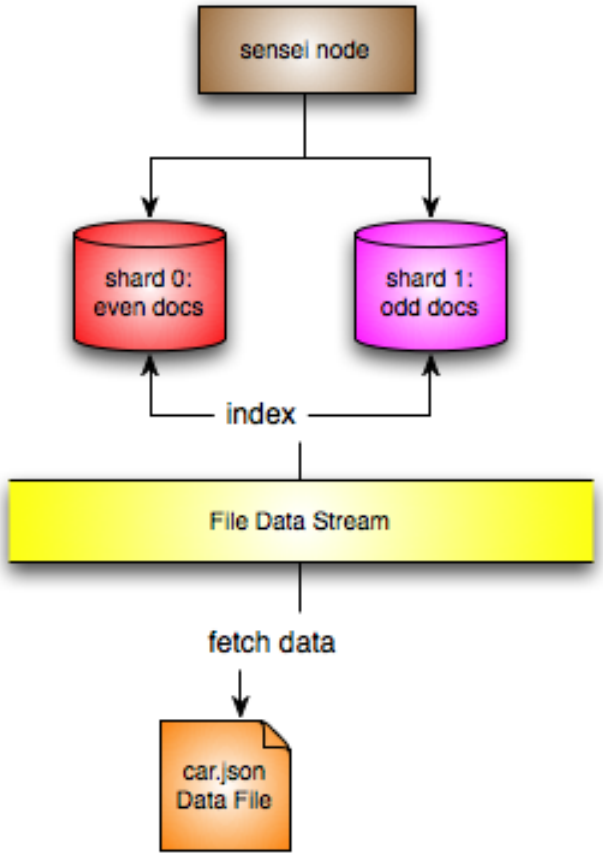


Figure 3.3. Reset Endpoint



Diagram

Figure 3.4. The Sensei Demo System



Chapter 4. Sensei Configuration

Overview

A Sensei node is configured by a set of files. These files describe a Sensei node in terms of data models, server configuration, indexing tuning parameters, customizations etc.

This document aims to describe how parts are pieced together via these configuration files.

Data Modeling

Data models are described in the `schema.xml` file (XSD definition: http://javasoze.github.com/sensei/schema/sensei-schema.xsd.)

The demo example can be found here: https://github.com/javasoze/sensei/blob/master/conf/schema.xml

The schema file is composed by 2 sections:

1. table
2. facets

Table Schema

A Sensei instance can be viewed as a giant table with many columns and many rows. The concept of such table directly correlates to that of traditional RDBMS's.

A table may have the following attributes:

- **uid** (mandatory) - defines the name of the primary key field. This must be of type `long`.
- **delete-field** (optional) - defines the field that would indicate a delete event (we will get back to this later).
- **skip-field** (optional) - defines the field that would indicate a skipping event (we will get back to this later).
- **src-data-store** (optional) - defines the format of how the stored field is saved.
- **compress-src-data** (optional) - defines if the stored field is compressed.

A table is also composed of a set of columns. Each column has a name and a type.

These are the supported types:

- **string** - value is a string, e.g. "`abc`"
- **int** - integer value
- **long** - long value
- **short** - short value
- **float** - a floating point value
- **double** - double value
- **char** - a character
- **date** - a date value, must be accompanied by a format string to be used to parse a data string
- **text** - a searchable text segment, standard Lucene indexing specification can also be specified here, e.g. `index="ANALYZED"`, `termvector="NO"` etc.

A column that is not of type "text" is considered a *meta* column. Any meta column can be specified to be either `single` (default) or `multi`. When a column is specified to be `multi`, e.g. `multi="true"`, this

means given a row, it can have more than 1 value for this column. A delimited string can be provided to help the indexer to parse the values (default delimiter is ",", for example, to specify a different delimiter to be ":", we can do this by `delimiter=":"`)

Here is an example of the table schema (see https://github.com/javasoze/sensei/blob/master/conf/schema.xml):

```
<table uid="id" delete-field="" skip-field="">
  <column name="color" type="string" />
  <column name="category" type="string" />
  <column name="city" type="string" />
  <column name="makemodel" type="string" />
  <column name="year" type="int" />
  <column name="price" type="float" />
  <column name="mileage" type="int" />
  <column name="tags" type="string" multi="true" delimiter=","/>
  <column name="contents" type="text" index="ANALYZED"
          store="NO" termvector="NO"/>
</table>
```

JSON

By default, data objects inserted into Sensei are JSON objects.

Example:

Given the following table definition:

```
<table uid="id">
  <column name="color" type="string" />
  <column name="year" type="int" />
  <column name=tag" type="string" multi="true" />
  <column name="description" type="text" index="ANALYZED" store="NO" />
</table>
```

The following table shows as an example how a json object is mapped into the table:

JSON object

```
{
  id:1
  color:"red",
  year:2000,
  tag:"cool,leather",
  description:"i love this car"
}
```

Table view

id	color	year	tag	description
1	red	2000	cool, leather	i love this car

Deletes

To deleted a row from Sensei, simply insert a data object with the specified delete-field set to true.

Example:

Given the table schema:

```
<table uid="id" delete-field="isDelete">
...
</table>
```

The following JSON object would delete the row where id=5:

```
{
  id:5,
  isDelete:"true"
}
```

Skips

In cases where runtime logic decides whether a data object should be skipped, the skip field can be useful.

Example:

Given the table schema:

```
<table uid="id" skip-field="isSkip">
...
</table>
```

The following JSON object would be skipped from indexing:

```
{
  id:7,
  isSkip:"true"
}
```

Facet Schema

The second section is the facet schema, which defines how columns can be queried.

If we think of the table section defines how data is added into Sensei, then the facet section describes how these data can be queried.

The facet sections is composed of a set of facet definitions.

A facet definition requires a name and a type.

Possible types:

- **simple**: simplest facet, 1 row = 1 discrete value
- **path**: hierarchical facet, e.g. a/b/c
- **range**: range facet, used to support range queries
- **multi**: 1 row = N discrete values
- **compact-multi**: similar to multi, but possible values are limited to 32
- **custom**: any user defined facet type (We will cover this in advanced section)

Example: https://github.com/javasoze/sensei/blob/master/conf/schema.xml

Optional Attributes

Column

The column attribute references the column names defined in the table section. By default, the value of the `name` attribute is used.

This can be useful if you want to name the facet name to be different from the defined column name, or if you want to have multiple facets defined on the same column.

Depends

This is a comma delimited string denoting a set of facet names this facet is to be depended on.

When `depends` is specified, Sensei guarantees the depending facets are loaded before this facet.

This is also how Composite Facets are constructed. (Another advanced topic).

Dynamic

Dynamic facets are useful when data layout is not known until query time.

Some examples:

- searcher's social network
- dynamic time ranges from when the search request is issued

This is another advanced topic to be discussed later.

Parameters

A facet can be configured via a list of parameters. Parameters are needed for a facet under some situations, for examples:

- For path facets, separator strings can be configured
- For range facets, predefined ranges can be configured

The parameters can be specified via element `params`, which contains a list of elements called `param`. For each `param`, two attributes can be specified: `name` and `value`.

How parameters are interpreted is dependent on the facet type.

Here is an example of a facet with a list of predefined ranges:

```
<facet name="year" type="range">
  <params>
    <param name="range" value="1993-1994"/>
    <param name="range" value="1995-1996"/>
    <param name="range" value="1997-1998"/>
    <param name="range" value="1999-2000"/>
    <param name="range" value="2001-2002"/>
  </params>
<facet>
```

Customized Facets

We understand we cannot possibly cover all use cases using a short list of predefined facet handlers. It is necessary to allow users to define their own customized facets for different reasons.

If a customized facet handler is required for a column (or multiple columns), you can set the facet `type` to `custom`, and declare a bean for the facet handler in file `custom-facets.xml`.

For example, if a customized facet called `time` is declared in `schema.xml` like this:

```
<facet name="time" type="custom" dynamic="false"/>
```

and the implementation of the facet handler is in class `com.example.facets.TimeFacetHandler`, then you should include the following line in file `custom-facets.xml` [1]:

```
<bean id="time" class="com.example.facets.TimeFacetHandler"/>
```

The id of the bean should match the name of the facet.

System Configuration

The Sensei system is configured via the `sensei.properties`, which consists of the following sections:

1. **server**: e.g. port to listen on, rpc paramters etc.
2. **cluster**: cluster manager, sharding, request routing etc.
3. **indexing**: data interpretation, tokenization, indexer type etc.
4. **broker**: e.g. entry into Sensei system
5. **plugins**: e.g. customized facet handlers

Below is the configuration file for the demo (available from https://github.com/javasoze/sensei/blob/master/conf/sensei.properties

```
# sensei node parameters
sensei.node.id=1
sensei.node.partitions=0,1

# sensei network server parameters
sensei.server.port=1234
sensei.server.requestThreadCorePoolSize=20
sensei.server.requestThreadMaxPoolSize=70
sensei.server.requestThreadKeepAliveTimeSecs=300

# sensei cluster parameters
sensei.cluster.name=sensei
sensei.cluster.url=localhost:2181
sensei.cluster.timeout=30000

# sensei indexing parameters
sensei.index.directory = index/cardata

sensei.index.batchSize = 10000
sensei.index.batchDelay = 300000
sensei.index.maxBatchSize = 10000
sensei.index.realtime = true
sensei.index.freshness = 10000
```

[1] Here we assume that the time facet handler does not take any arguments.

```
# index manager parameters

sensei.index.manager.default.maxpartition.id = 1
sensei.index.manager.default.type = file
sensei.index.manager.default.file.path = data/cars.json

# plugins: from plugins.xml

# analyzer, default: StandardAnalyzer
# sensei.index.analyzer = myanalyzer

# similarity, default: DefaultSimilarity
# sensei.index.similarity = mysimilarity

# indexer type, zoie/hourglass/<custom name>

sensei.indexer.type=zoie

#extra parameters for hourglass

#sensei.indexer.hourglass.schedule

# retention
#sensei.indexer.hourglass.timethreshold

# frequency for a roll, minute/hour/day
#sensei.indexer.hourglass.frequency

# sensei
# version comparator, default: ZoieConfig.DefaultVersionComparator
# sensei.version.comparator = myVersionComparator

# extra services
sensei.plugin.services =

# broker properties
sensei.broker.port = 8080
sensei.broker.minThread = 50
sensei.broker.maxThread = 100
sensei.broker.maxWaittime = 2000

sensei.broker.webapp.path=src/main/webapp
sensei.search.cluster.name = sensei
sensei.search.cluster.zookeeper.url = localhost:2181
sensei.search.cluster.zookeeper.conn.timeout = 30000
sensei.search.cluster.network.conn.timeout = 1000
sensei.search.cluster.network.write.timeout = 150
sensei.search.cluster.network.max.conn.per.node = 5
sensei.search.cluster.network.stale.timeout.mins = 10
sensei.search.cluster.network.stale.cleanup.freq.mins = 10

# custom router factory
# sensei.search.router.factory = myRouterFactory
```

Server Properties

sensei.node.id
- Type: int
- Default: None (this is a required property)

This is the node ID of the Sensei node in a cluster.

sensei.node.partitions
- Type: comma separated integers or ranges
- Default: None (this is a required property)

This specifies the partitions IDs this the Sensei server is going to handle. Partition IDs can be given as either integer numbers or ranges, separated by commas. For example: `sensei.node.partitions=1,2,5-8` denotes that the Sensei server has six partitions: 1,2,5,6,7,8.

sensei.server.port
- Type: int
- Default: None (this is a required property)

This is the Sensei server port number.

sensei.server.requestThreadCorePoolSize
- Type: int
- Default: 20

This is the core size of thread pool used to execute requests.

sensei.server.requestThreadMaxPoolSize
- Type: int
- Default: 70

This is the maximum size of thread pool used to execute requests.

sensei.server.requestThreadKeepAliveTimeSecs
- Type: int
- Default: 300

This is the length of time in seconds to keep an idle request thread alive.

Cluster Properties

sensei.cluster.name
- Type: String
- Default: None (required)

This is the name of the Sensei server cluster.

sensei.cluster.url
- Type: String
- Default: None (required)

This is the URL to be used to connect to the Sensei cluster.

sensei.cluster.timeout
- Type: int
- Default: 300000

This is the session timeout value, in milliseconds, that is passed to ZooKeeper.

Indexing Properties

sensei.index.directory
- Type: String
- Default: None (required)

This is the directory used to save the index.

sensei.index.batchSize
- Type: int
- Default: 10000

This is the batch size used by Zoie to control the pace of data event consumption. This batch size is the *soft* size limit of each event batch. If the events come in too fast and the limit is already reached, then Zoie will block the incoming events until the number of buffered events drop below this limit after some of the events are sent to the background data consumer.

sensei.index.batchDelay
- Type: int
- Default: 300000

This is the maximum time to wait in milliseconds before flushing index events to disk. The default value is 5 minutes.

sensei.index.maxBatchSize
- Type: int
- Default: 10000

This is the maximum batch size.

sensei.index.realtime
- Type: boolean
- Default: true

This specifies whether the indexing mode is real-time.

sensei.index.freshness
- Type: long
- Default: 500

This controls the freshness of entries in the index reader cache.

sensei.indexer.type
- Type: String
- Default: None

This is the internal indexer type used by the Sensei cluster. Currently only two options are supported: `zoie` and `hourglass`. If `hourglass` is used, three more properties need to be set too:

- sensei.indexer.hourglass.schedule
- sensei.indexer.hourglass.timethreshold
- sensei.indexer.hourglass.frequency

sensei.indexer.hourglass.schedule

- Type: String
- Default: None

This is a string that specifies Hourglass rolling forward schedule. The format of this string is "`ss mm hh`", meaning at `hh:mm:ss time` of the day that we roll forward for *daily* rolling. If it is *hourly* rolling, we roll forward at `mm:ss` time of the hour. If it is *minutely* rolling, we roll forward at `ss` seond of the minute.

sensei.indexer.hourglass.trimthreshold

- Type: int
- Default: 14

This is the retention period for how long we are going to keep the events in the index. The unit is the rolling period.

sensei.indexer.hourglass.frequency

- Type: String
- Default: "`day`"

This is the rolling forward frequency. It has to be one of the following three values:

- `day`
- `hour`
- `minute`

Broker Properties

sensei.broker.port

- Type: int
- Default: None

This is the port number of the Sensei broker.

sensei.broker.webapp.path

- Type: String
- Default: None

This is the resource base of the broker web application.

sensei.broker.minThread

- Type: int
- Default: 20

This is the core size of thread pool used by the broker to execute requests.

sensei.broker.maxThread

- Type: int
- Default: 50

This is the maximum size of thread pool used by the broker to execute requests.

sensei.broker.maxWaittime
- Type: int
- Default: 2000

This is the maximum idle time in milliseconds for a thread on brokers. Threads that are idle for longer than this period may be stopped.

Client Properties

sensei.search.cluster.zookeeper.url
- Type: String
- Default: None

This is the ZooKeeper URL for the network client of a Sensei cluster.

sensei.search.cluster.name
- Type: String
- Default: None

This is the Sensei cluster name, the service name for the network client.

sensei.search.cluster.zookeeper.conn.timeout
- Type: int
- Default: 10000

This is the ZooKeeper network client session timeout value in milliseconds.

sensei.search.cluster.network.conn.timeout
- Type: int
- Default: 1000

This is the maximum number of milliseconds to allow a connection attempt to take.

sensei.search.cluster.network.write.timeout
- Type: int
- Default: 150

This is the number of milliseconds a request can be queued for write before it is considered stale.

sensei.search.cluster.network.max.conn.per.node
- Type: int
- Default: 5

This is the maximum number of open connections to a node.

sensei.search.cluster.network.stale.timeout.mins
- Type: int
- Default: 10

This is the number of minutes to keep a request that is waiting for a response.

sensei.search.cluster.network.stale.cleanup.freq.mins
- Type: int
- Default: 10

This is the frequency to clean up stale requests.

Plug-in Properties

sensei.index.analyzer
- Type: String
- Default: ""

This specifies the bean ID of the analyzer plug-in for analyzing text. If not specified, `org.apache.lucene.analysis.standard.StandardAnalyzer` will be used.

sensei.index.similarity
- Type: String
- Default: ""

This specifies the bean ID of similarity plug-in for Lucene scoring. If not specified, `org.apache.lucene.search.DefaultSimilarity` is used.

sensei.index.interpreter
- Type: String
- Default: ""

This specifies the bean ID of the interpretor of Zoie indexables. If not specified, `com.sensei.indexing.api.DefaultJsonSchemaInterpreter` is to be used.

sensei.index.manager.default.type
- Type: String
- Default: None

This is the type of the index manager data provider that will be used by Sensei. Currently the following types are supported:
- `file`
- `kafka`
- `custom`

If a customized index manager is provided, property `sensei.index.manager.default.custom` and `sensei.index.manager.default.filter` should be set too.

sensei.index.manager.default.custom
- Type: String
- Default: None

This is the bean ID of the customized data provider used by user provided index manager.

sensei.index.manager.default.filter
- Type: String
- Default: None

This is the bean ID of the JSON filter used by user provided index manager.

Extensions

(TO BE FINISHED)