Sensei

A Distributed Elastic Real-Time Searchable Database

Sensei Development Team

Sensei: A Distributed Elastic Real-Time Searchable Database

Sensei Development Team

Table of Contents

List of Figures

Chapter 1. Introduction

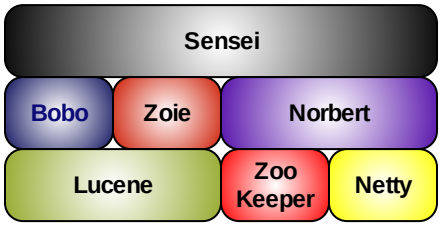
Sensei is an open-source, real-time, full-text searchable distributed database that is designed to handle the following type of queries:

```
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
```

Sensei is written in Java and is built on top of several other open-source software systems (see Figure 1.1, “Sensei and Its Foundation”):

- Bobo (http://sna-projects.com/bobo/): a faceted search implementation written in Java, using Lucene as the underline search and indexing engine.
- Zoie (http://sna-projects.com/zoie/): a real-time search and indexing system built on Lucene.
- Apache Lucene (http://lucene.apache.org/): a high-performance, full-featured text search engine library written entirely in Java.
- Norbert (http://sna-projects.com/norbert/): a library that provides easy cluster management and workload distribution. Norbert is built on ZooKeeper (http://zookeeper.apache.org/) and Netty (http://www.jboss.org/netty)

Figure 1.1. Sensei and Its Foundation



1.1. Design Considerations

As another NoSQL system (http://nosql-database.org/), Sensei is designed and built with the following 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.

1.2. 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

1.3. Architecture

At a high level, a Sensei system consists of two parts: a cluster of Sensei *servers* (a.k.a. *search nodes*) and a cluster of Sensei *brokers*.

1. The cluster of Sensei servers:

   Each server covers one or more *partitions* (or *shards*) of the entire index space, and is responsible for real-time indexing and searching on the partition(s) belonging to the node.

   > **Partition and Shard**
   >
   > A partition or shard is just a slice of the index served by the entire Sense cluster. Each partition may have multiple replicas handled by different Sensei search nodes. How data is partitioned (or sharded) is up to the business logic of the application, and the sharding strategy can be passed to Sensei as a plug-in.

2. The cluster of Sensei brokers:

   Brokers receive search requests from clients, pass them to selected servers in the Sensei search cluster, and then merge/return search results back to the clients.

Sensei brokers are relatively simple and lightweight. Most of the work is done at Sensei servers. A Sensei server plays two roles: indexer and searcher, both of which are implemented by Zoie and Bobo embedded in the node.

Conceptually, a Sensei node contains the following four components:

1. Data provider

   This is the component responsible for getting data from external sources and passing them to the indexer. Several built-in data providers are available in Sensei, and they can be used to get data from common data sources:
   - Line-based text file containing JSON objects.
   - Streaming JSON objects.
   - Kafka (http://sna-projects.com/kafka/)
   - JMS
   - JDBC

   For each data provider, a filter can be plugged in to convert the original source data into the format defined by the table schema. The output of the filter should be of JSON format, which is the internal format used by Sensei.

2. Indexing manager

   Indexing manager acts as the bridge between the data provider and the Zoie system. It is responsible for passing data from the data provider to Zoie, controlling the pace of data consumption, and maintaining the index versions on the Sensei node.

3. Zoie system

   This is the underlying system powering real-time indexing and search. Two types of Zoie systems are supported by Sensei today: regular Zoie and Hourglass.

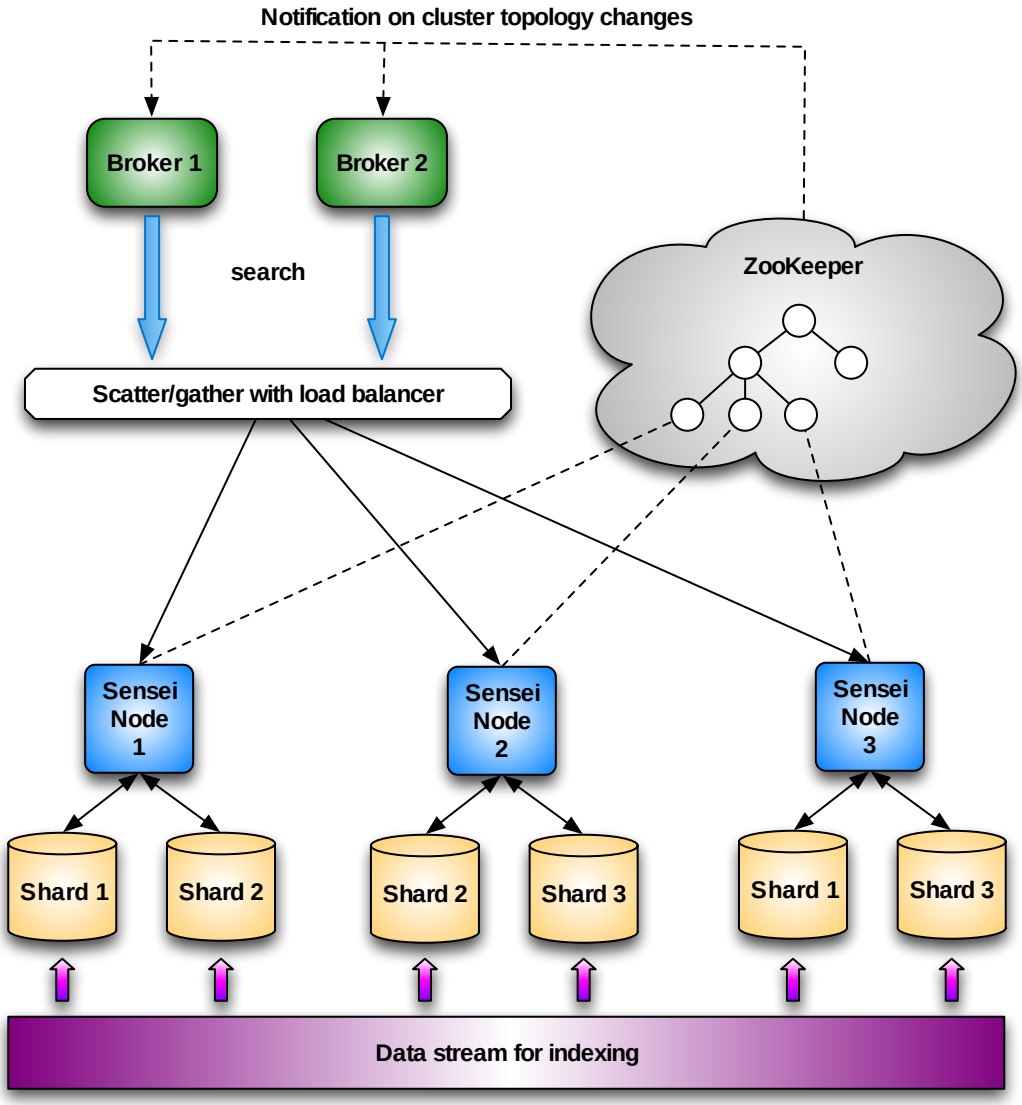
   Hourglass (http://linkedin.jira.com/wiki/display/ZOIE/HourGlass+-+Forward-Rolling+Indexing) is a forward-rolling, append-only indexing system based on Zoie. It is used to power LinkedIn Signal (http://www.linkedin.com/signal).

4. Facet handlers

   Facet handlers are a key component in a Sensei server. They are the way we do faceted searches with Bobo. Without facet handlers, none of the column-based queries can be done in Sensei.

1.4. Architectural Diagram

Figure 1.2. Sensei Architectural Diagram



Chapter 2. Getting Started

2.1. Overview

- Real-time indexing/searching
- Cluster management
- Automatic data partitioning
- Support for structured and faceted search

2.2. Prerequisites

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

2.3. Embedded Technologies

- Bobo (http://sna-projects.com/bobo/)
- Zoie (http://sna-projects.com/zoie/)
- Apache Lucene (http://lucene.apache.org/)
- Norbert (http://sna-projects.com/norbert/)
- ZooKeeper (http://zookeeper.apache.org/)
- Spring (http://www.springsource.com/)

2.4. Details

2.4.1. Building Sensei

Getting the Sensei source code and building the entire system is straightforward. Only two commands are needed:

1. Checking out the source code from trunk:

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

2. Building Sensei using `ant`:

```
$ ant
```

2.4.2. Starting ZooKeeper

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

You may download ZooKeeper 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.

2.4.3. Starting Sensei Nodes

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

Here is 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.

2.4.4. Web Application and RESTful End-Point

When developing applications using Sensei, we found it convenient to have a RESTful end-point that provides data in JSON format. Having a RESTful end-point allows one to query the Sensei system in an ad-hoc way, and having a JSON formatted output provides a way to investigate the result set without having the need to depend on jars or any other types of code binding.

When a Sensei node is started, a RESTful end-point along with a web interactive client would be started as well:

- RESTful end-point: http://localhost:8080/sensei?q=
- Web client: http://localhost:8080

2.4.5. Starting Clients

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 the properties)

Type help to see command list:

```
$ bin/sensei-client.sh client-conf
> help
help - prints this message
exit - quits
info - prints system 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 indicates a not
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

3.1. 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/`

3.2. Run The Demo

Build:

```
$ ant
```

Make sure ZooKeeper is running:

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

Run:

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

3.3. URLs

- Sensei Web Client: http://localhost:8080/
- Demo: http://localhost:8080/
- RESTful End-Point: http://localhost:8080/sensei

3.4. Screenshots

Figure 3.1. Sensei Web Client

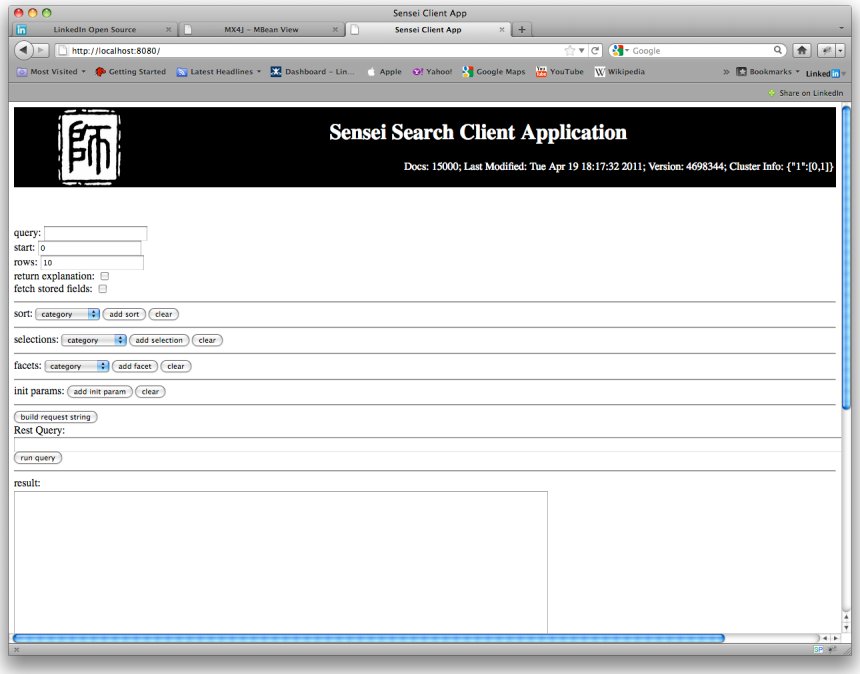


Figure 3.2. Demo

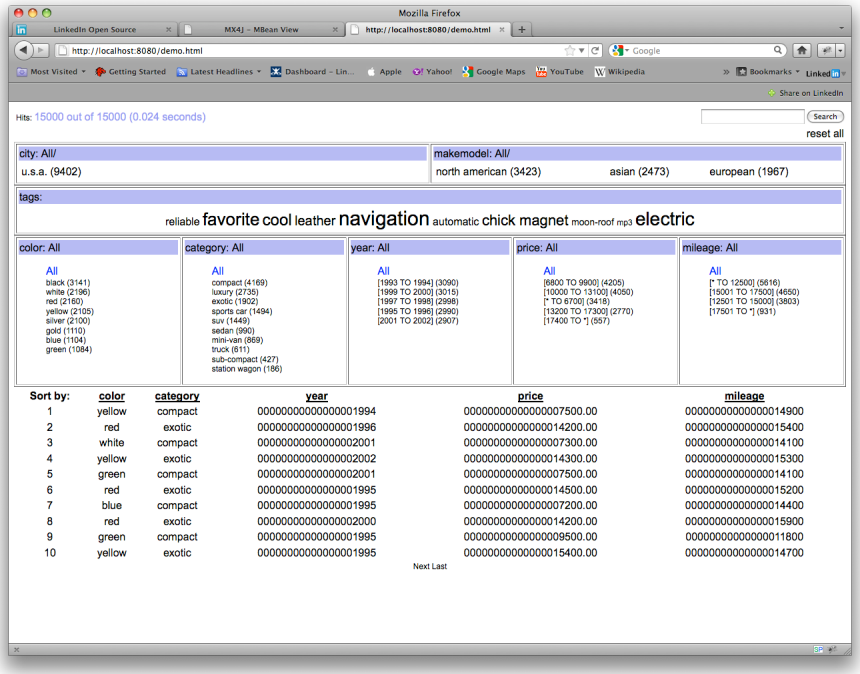
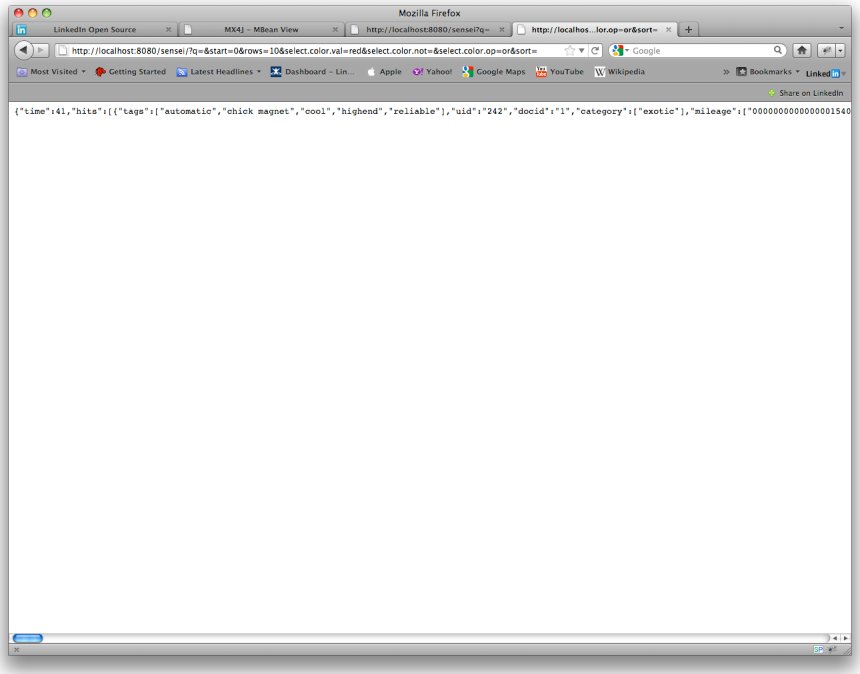
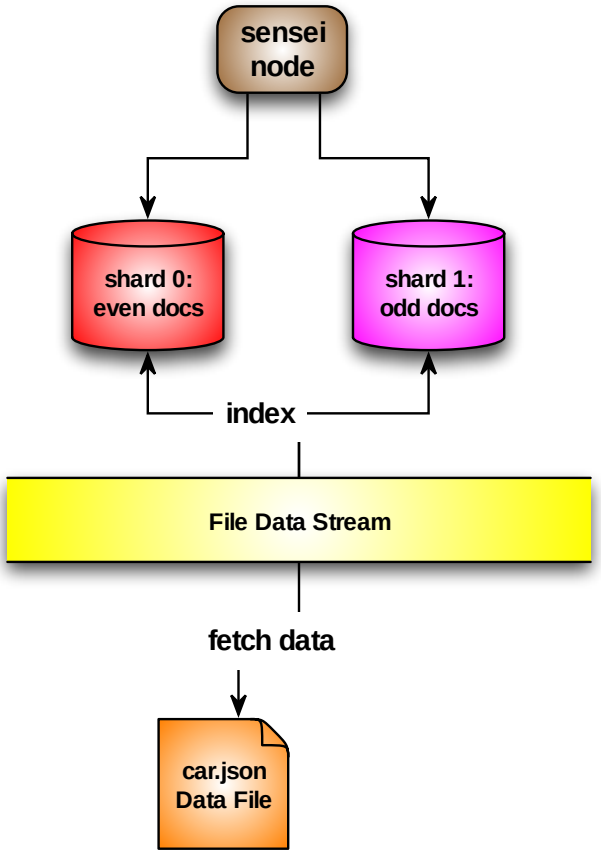


Figure 3.3. RESTful End-Point



3.5. Diagram

Figure 3.4. The Sensei Demo System



3.6. Demo Configuration

The configuration for the demo application can be found at: https://github.com/javasoze/sensei/tree/master/conf, which contains the following files:

- `sensei.properties`: main configuration file, describes the overall configuration of the system (we will dig into that in detail later)
- `schema.xml`: this describes the data model and the faceting behavior of the application
- `custom-facets.xml`: Spring file for adding custom facets, while Sensei comes with a set of facet types out of the box, this allows user-defined facets to be plugged into the application
- `plugins.xml`: any other plug-ins, e.g. Analyzer, Similarity, Interpreter etc.

3.6.1. Data Model

The data model is described in `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>

<facets>
  <facet name="color" type="simple" depends=""/>
  <facet name="category" type="simple" />
  <facet name="city" type="path">
    <params>
      <param name="separator" value="/" />
    </params>
  </facet>
  <facet name="makemodel" type="path" />
  <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>
  <facet name="mileage" type="range">
    <params>
      <param name="range" value="*-12500" />
      <param name="range" value="12501-15000" />
      <param name="range" value="15001-17500" />
      <param name="range" value="17501-*" />
    </params>
  </facet>
  <facet name="price" type="range">
       <params>
         <param name="range" value="*,6700" />
         <param name="range" value="6800,9900" />
         <param name="range" value="10000,13100" />
         <param name="range" value="13200,17300" />
         <param name="range" value="17400,*" />
       </params>
  </facet>
  <facet name="tags" type="multi" />
</facets>
```

This corresponds to the following data table:

	color	**category**	**city**	**makemodel**	**year**	**price**	**mileage**	**tags**	**contents**
meta/ structured	yes	yes	yes	yes	yes	yes	yes	yes	no
type	string	string	string	string	int	float	int	string	text
facet	simple	simple	path	path	range	range	range	multi	N/A

Chapter 4. Sensei Configuration

4.1. 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 chapter aims to describe how different parts are pieced together via these configuration files.

4.2. Data Modeling

Data models are described in the `schema.xml` file. The XSD definition of this XML file can be found from http://javasoze.github.com/sensei/schema/sensei-schema.xsd.

The schema file is composed by 2 sections:

1. Table schema
2. Facet schema

4.2.1. 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 source data is saved. Currently the only supported value is "`lucene`".
- **src-data-field** (optional) - specifies the field name used to keep the original source data value.

Note

If this attribute is not specified, the default value is set to "`src_data`". If this field is not set by the data source filter, the string representation of the original data source is saved in this field by default. If part of the source data or a modified version of the source data is to be saved in the index, then you need to set this field using the value you prefer in the data source filter.

- **compress-src-data** (optional) - defines if the source data is compressed.

A table is also composed of a set of columns. Each column has a name and a type. Below is the list of 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, which must be accompanied by a format string to be used to parse a date string

- **text** - a searchable text segment, standard Lucene indexing specification can also be specified here, e.g. `index="ANALYZED"`, `termvector="NO"`.

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"`, it means that, given a row, the column can have more than one value. A delimited string can be provided to help the indexer parse the values (default delimiter is ","). To specify a different delimiter, say ":", we can simply set `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>
```

4.2.1.1. 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

4.2.1.2. Deletes

To delete 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"
}
```

4.2.1.3. 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"
}
```

4.2.1.4. Source JSON

For many cases, you may want to save the original source data from which we extract all the fields into the index. You can do this by setting the attributes **src-data-store** and **src-data-field**.

4.2.2. 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

4.2.2.1. Optional Attributes

4.2.2.1.1. 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.

4.2.2.1.2. Depends

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

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

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

4.2.2.1.3. 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.

4.2.2.2. Parameters

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

- 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 need to be specified: `name` and `value`.

How parameters are interpreted and used 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>
```

4.2.2.3. 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.

4.3. System Configuration

A Sensei node is configured via the `sensei.properties`, which uses the format supported by Apache Commons Configuration (http:/commons.apache.org/). This file consists of the following five parts:

1. **server**: port to listen on, rpc parameters, etc.
2. **cluster**: cluster manager, sharding, request routing, etc.
3. **indexing**: data interpretation, tokenization, indexer type, etc.
4. **broker and client**: 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 ❶
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 ❷
sensei.cluster.name=sensei
sensei.cluster.url=localhost:2181
sensei.cluster.timeout=30000

# sensei indexing parameters ❸
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

# 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 ❹
```

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

```
# 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 ❺
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
```

❶ This lines starts the server configurations.
❷ This lines starts the cluster configurations.
❸ This lines starts the indexing configurations.
❹ This lines starts the plugins configurations.
❺ This lines starts the broker and client configurations.

In the following sections, we are going to explain every configuration property in each part: what the property type is, whether it is required, what the default value is, and how it is used, etc.

4.3.1. Server Properties

sensei.node.id

- Type: int
- Required: Yes
- Default: None

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

sensei.node.partitions
- Type: String (comma separated integers or ranges)
- Required: Yes
- Default: None

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, the following line denotes that the Sensei server has six partitions: 1,4,5,6,7,10.

```
sensei.node.partitions=1,4-7,10
```

sensei.server.port
- Type: int
- Required: Yes
- Default: None

This is the Sensei server port number.

sensei.server.requestThreadCorePoolSize
- Type: int
- Required: No
- Default: 20

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

sensei.server.requestThreadKeepAliveTimeSecs
- Type: int
- Required: No
- Default: 300

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

sensei.server.requestThreadMaxPoolSize
- Type: int
- Required: No
- Default: 70

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

4.3.2. Cluster Properties

sensei.cluster.name
- Type: String
- Required: Yes
- Default: None

This is the name of the Sensei server cluster.

sensei.cluster.timeout
- Type: int
- Required: No
- Default: 300000

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

sensei.cluster.url
- Type: String

- Required: Yes
- Default: None

This is the ZooKeeper URL for the Sensei cluster.

4.3.3. Indexing Properties

sensei.index.analyzer
See **sensei.index.analyzer** in Section 4.3.5, “Plug-in Properties”.

sensei.index.batchDelay
- Type: int
- Required: No
- Default: 300000

This is the maximum time to wait in milliseconds before flushing index events to disk. The default value is 300000 (i.e. 5 minutes).

sensei.index.batchSize
- Type: int
- Required: No
- 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.directory
- Type: String
- Required: Yes
- Default: None

This is the directory used to save the index.

sensei.index.freshness
- Type: long
- Required: No
- Default: 500

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

sensei.index.interpreter
See **sensei.index.interpreter** in Section 4.3.5, “Plug-in Properties”.

sensei.index.manager
See **sensei.index.manager** in Section 4.3.5, “Plug-in Properties”.

sensei.index.manager.default.maxpartition.id
- Type: int
- Required: Yes, if the default indexing manager is chosen; No, otherwise.
- Default: None

This is the maximum partition ID number served by this Sensei cluster if the default Sensei indexing manager is used.

Warning

This property is different from the total number of partitions in a Sensei cluster. For example, if a cluster contains 4 partitions, 0, 1, 2, and 3, then **sensei.index.manager.default.maxpartition.id** should be set to 3.

sensei.index.manager.default.shardingStrategy
See **sensei.index.manager.default.shardingStrategy** in Section 4.3.5, “Plug-in Properties”.

sensei.index.manager.default.type
See **sensei.index.manager.default.type** in Section 4.3.5, “Plug-in Properties”.

sensei.index.maxBatchSize
- Type: int
- Required: No
- Default: 10000

This is the maximum batch size.

sensei.index.realtime
- Type: boolean
- Required: No
- Default: true

This specifies whether the indexing mode is real-time or not.

sensei.index.similarity
See **sensei.index.similarity** in Section 4.3.5, “Plug-in Properties”.

sensei.indexer.type
- Type: String
- Required: Yes
- 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:
1. **sensei.indexer.hourglass.schedule**
2. **sensei.indexer.hourglass.timethreshold**
3. **sensei.indexer.hourglass.frequency**

sensei.indexer.hourglass.frequency
- Type: String
- Required: No
- Default: "`day`"

This is the rolling forward frequency. It has to be one of the following three values:
- `day`
- `hour`
- `minute`

sensei.indexer.hourglass.schedule
- Type: String
- Required: Yes, if property **sensei.indexer.type** is set to "`hourglass`"; No, otherwise.
- 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
- Required: No
- 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.

4.3.4. Broker and Client Properties

sensei.broker.maxThread
- Type: int
- Required: No
- Default: 50

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

sensei.broker.maxWaittime
- Type: int
- Required: No
- Default: 2000

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

sensei.broker.minThread
- Type: int
- Required: No
- Default: 20

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

sensei.broker.port
- Type: int
- Required: Yes
- Default: None

This is the port number of the Sensei broker.

sensei.broker.webapp.path
- Type: String
- Required: Yes
- Default: None

This is the resource base of the broker web application.

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

This is the ZooKeeper URL for the Sensei search cluster that a broker talks to.

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

This is the Sensei cluster name, i.e. the service name for the network clients and brokers.

sensei.search.cluster.zookeeper.conn.timeout
- Type: int
- Required: No
- Default: 10000

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

sensei.search.cluster.network.conn.timeout
- Type: int
- Required: No
- Default: 1000

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

sensei.search.cluster.network.write.timeout
- Type: int
- Required: No
- 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
- Required: No
- Default: 5

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

sensei.search.cluster.network.stale.timeout.mins
- Type: int
- Required: No
- 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
- Required: No
- Default: 10

This is the frequency to clean up stale requests.

4.3.5. Plug-in Properties

sensei.index.analyzer
- Type: String
- Required: No
- 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
- Required: No
- Default: ""

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

sensei.version.comparator

- Type: String
- Required: No
- Default: ""

This specifies the bean ID of version comparator plug-in to be used by Zoie. If not specified, Zoie's default version comparator is used.

sensei.index.interpreter

- Type: String
- Required: No
- Default: ""

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

sensei.index.manager

- Type: String
- Required: No
- Default: ""

This specifies the bean ID of the indexing manager object implementing `com.sensei.search.nodes.SenseiIndexingManager`. If not specified, `com.sensei.indexing.api.DefaultStreamingIndexingManager` is used.

sensei.index.manager.default.type

- Type: String
- Required: Yes if **sensei.index.manager** is not specified, i.e. the default indexing manager is used.
- Default: None

This specifies the type of the data provider that will be used by the default indexing manager. The value identifies the bean ID of the object of a data provider builder:

```
com.sensei.indexing.api.DataProviderFactoryRegistry.DataProviderBuilder
```

Several built-in data providers are provided by Sensei, but you can always define your own version based on your need. No matter a built-in data provider or a custom data provider is used, additional parameters can be specified under names with prefix **sensei.index.manager.default.<data-provider-type>**.

Currently the following built-in data provider types are supported:

- `file`:

  This type of data provider takes a regular text file as the input. Each line in the file contains a data entry in JSON format.

Only one property needs to be set for this type of data providers. See Section 4.3.5.1, “File Data Provider Properties”

- `kafka`:

  This type of data provider takes Kafka messages as input.

  See Section 4.3.5.2, “Kafka Data Provider Properties” for additional property information.

- `jms`:

  This type of data provider takes JMS (Java Messages Service) messages as input. The publish-and-subscribe messaging model is used by Sensei, so parameters like topic need to be provided.

  See Section 4.3.5.3, “JMS Data Provider Properties” for additional property information.

- `jdbc`:

  This type of data provider takes JDBC data as input.

  See Section 4.3.5.4, “JDBC Data Provider Properties” for additional property information.

sensei.index.manager.default.<data-provider-type>.filter

- Type: String
- Required: No
- Default: None

This is the bean ID of `com.sensei.indexing.api.DataSourceFilter` object. No matter what data provider the indexing managers uses, a filter can be plugged in to get the original source data converted to the JSON format defined by the table schema. If the input data is already in the right format, then this filter is not needed.

sensei.index.manager.default.shardingStrategy

- Type: String
- Required: No
- Default: ""

This is the bean ID of the sharding strategy.

4.3.5.1. File Data Provider Properties

For `file` data providers, the following property has to be specified:

sensei.index.manager.default.file.path

- Type: String
- Required: Yes
- Default: None

This is the path to the input data file.

4.3.5.2. Kafka Data Provider Properties

For `kafka` data providers, the following properties should/can be specified: [2]

[2] These properties are basically the parameters needed by the Kafka consumer API. The Simple Consumer API from Kafka is used by Sensei.

sensei.index.manager.default.kafka.batchsize
- Type: String
- Required: Yes
- Default: None

This is the batch size for each pull request.

sensei.index.manager.default.kafka.host
- Type: String
- Required: Yes
- Default: None

This is the host name of the Kafka server.

sensei.index.manager.default.kafka.port
- Type: int
- Required: Yes
- Default: None

This is the port number on which the Kafka server is listening for connections.

sensei.index.manager.default.kafka.timeout
- Type: int
- Required: Yes
- Default: 10000

This is the socket timeout in milliseconds.

sensei.index.manager.default.kafka.topic
- Type: String
- Required: Yes
- Default: None

The topic of the messages to be fetched.

4.3.5.3. JMS Data Provider Properties

For `jms` data providers, the following properties should/can be specified:

sensei.index.manager.default.jms.clientId
- Type: String
- Required: Yes
- Default: None

This is the client identifier used to connect to the JMS provider.

sensei.index.manager.default.jms.topic
- Type: String
- Required: Yes
- Default: None

This is the topic name that the JMS client subscribes to.

sensei.index.manager.default.jms.topicFactory
- Type: String
- Required: Yes

- Default: None

This is the bean ID of the `proj.zoie.dataprovider.jms.TopicFactory` object. This object is used to generate a topic object based on the given topic name.

sensei.index.manager.default.jms.connectionFactory
- Type: String
- Required: Yes
- Default: None

This is the bean ID of the `javax.jms.TopicConnectionFactory` object, which is used by the JMS client to create a `javax.jms.TopicConnection` object with the JMS provider.

4.3.5.4. JDBC Data Provider Properties

For `jdbc` data providers, the following properties should/can be specified:

sensei.index.manager.default.jdbc.adaptor
- Type: String
- Required: Yes
- Default: None

This is the bean ID of the `com.sensei.indexing.api.jdbc.SenseiJDBCAdaptor` object. This object is used to build a `proj.zoie.dataprovider.jdbc.PreparedStatementBuilder` object, which is required by `proj.zoie.dataprovider.jdbc.JDBCStreamDataProvider`.

sensei.index.manager.default.jms.driver
- Type: String
- Required: Yes
- Default: None

This is the class name of the JDBC driver that you want to use.

sensei.index.manager.default.jms.password
- Type: String
- Required: Yes
- Default: None

This is the password for the user name that you use to connect to the database.

sensei.index.manager.default.jms.username
- Type: String
- Required: Yes
- Default: None

This is the user name that you use to connect to the database.

Chapter 5. FAQ

5.1. About Sensei

5.1.1. Why is it called Sensei?

(To Be Finished)

5.2. Sensei Configuration

5.2.1. aaa

(To Be Finished)

5.3. Problems Running Sensei

5.3.1. aaa

(To Be Finished)

Index