

**COSC 416**  
***NoSQL Databases***

***SenseiDB***

**Alex YellowShoes, Jamie McKee, Luke  
Warkotsch, Ryan Trenholm  
University of British Columbia Okanagan**



# SenseiDB

**Sensei** is an open-source, distributed, real-time, semi-structured data system that acts as both a search engine and a database.

Designed to query and navigate through documents.

- ◆ Text and unstructured sections
- ◆ Meta information and well-formed structured sections

Powers the **LinkedIn homepage** and **LinkedIn Signal** application.

## Sensei the Name

Sensei (先生) means teacher or professor in Japanese (<http://en.wikipedia.org/wiki/Sensei>).

It shares the same pronunciation and writing with the Chinese word that has the same meaning. This name indicates that the system can be used in place of Oracle database in many applications.

Is winning  
business  
strategy!



# *What is SenseiDB?*

---

Key-value store database that offers

- ◆ Full-text search
- ◆ Fast key-value lookup
- ◆ Fast real-time updates
- ◆ Structured and faceted search

Important things to note:

- ◆ Sensei runs a single **node** (Java process) that performs indexing work and handles query requests
- ◆ That node can query over N partitions of data
- ◆ Sensei only has a single **index** and doesn't support **JOINS**
- ◆ Data objects in Sensei are JSON objects

# RDBMS vs. SenseiDB

---

## RDBMS

- ◆ Scales vertically
- ◆ Strong ACID guarantee
- ◆ Relational model support
- ◆ Performance cost for full-text integration
- ◆ High query latency for large datasets
- ◆ Indexes have to be built for sorting

## Sensei

- ◆ Scales horizontally
- ◆ Relaxed Consistency but high Durability guarantee
- ◆ Atomicity and Isolation handled by data producer
- ◆ Deep full-text integration
- ◆ Low query latency for large datasets
- ◆ Dynamic sorting since index already built in



# SenseiDB Schemas

---

**Table schema** defines *how data is stored* in Sensei.

- ◆ String, int, long, short, float, double, char, date
- ◆ Text is a searchable text segment!
- ◆ UID : long (mandatory; defines name of primary key field)

**Facet schema** describes *how we can query columns of data*.

- ◆ Simple : row has single discrete value
- ◆ Range : supports range queries
- ◆ Multi : row has N discrete values
- ◆ TimeRange : search based on time column value within range
- ◆ And many more!

# Create table in Sensei

---

The table is defined in the `schema.xml` file

## Example:

```
<table uid="id" delete-field="isDelete">
  <column name="gid" type="int" />
  <column name="gname" type="string" />
  <column name="pubname" type="string" />
  <column name="releasedate" type="date" />
  <column name="maxplayers" type="int" />
  <column name="rating" type="float" />
  <column name="genre" type="string" multi="true" delimiter=","/>
  <column name="description" type="text" index="ANALYZED" store="NO"
termvector="NO"/>
</table>
```

# Create facet in Sensei

---

Facets are also defined in the `schema.xml` file

## Example:

```
<facets>
  <facet name="gid" type="int" />
  <facet name="gname" type="simple" />
  <facet name="pubname" type="simple" />
  <facet name="releasedate" type="simple" />
  <facet name="maxplayers" type="simple" />
  <facet name="rating" type="range">
    <params>
      <param name="range" value="0-1" />
      <param name="range" value="2-3" />
      <param name="range" value="4-5" />
    </params>
  </facet>
  <facet name="genre" type="multi" />
  <facet name="description" type="simple" />
</facets>
```



# Browse Query Language (BQL)

---

Sensei defines a SQL-variant query language called *Browse Query Language (BQL)*.

- ◆ **NOTE:** *BQL only supports **SELECT** statements!*
- ◆ **FROM** clause (and index name) is actually optional

**SELECT** statement schema:

```
SELECT <select_list>
[ FROM      <index> ]
[ WHERE     <search_expression> ]
[ GIVEN FACET PARAM   <facet_param_list> ]
( <order_by_clause>
  | <group_by_clause>
  | <limit_clause>
  | <fetching_stored_clause>
) *
```



# BQL Predicates

---

Many familiar predicate options for the **WHERE** clause

◆ **IN, CONTAINS ALL, =, <>, >, <, >=, <=, BETWEEN, LIKE, ORDER BY, LIMIT, GROUP BY**

Full text search on one or more (string) columns

◆ e.g. **MATCH** (pubname, genre) **AGAINST** ("\*ac\*")

Full text search on the *contents* within a column

◆ e.g. **QUERY IS** "cool AND (Mario OR Legend)"

# ***BQL Predicates cont.***

---

BQL has several Time-based predicates for columns that contain a timestamp

- ◆ e.g. time **IN LAST** 2 hours 10 mins
- ◆ e.g. time **SINCE** 2 weeks **AGO**
- ◆ e.g. time **AFTER** 2013-01-30 15:30:00
- ◆ e.g. time **BEFORE** 2012-12-21 15:30:00
- ◆ e.g. time **NOT BEFORE** 2 weeks **AGO**

Example:

- ◆ `SELECT gname WHERE releasedate IN LAST 2 YEARS;`



# BQL Faceted Search

Supports faceted search to retrieve additional information along with the search results

◆ BROWSE BY <facet\_param\_list>

Example:

```
SELECT gname BROWSE BY genre;
```

| gname                          | genre                 |
|--------------------------------|-----------------------|
| 007: The World is not Enough   | Dance (85)            |
| 50 Cent: Bulletproof           | RTS (85)              |
| AMF Bowling Pinbusters!        | Puzzle (80)           |
| Ace Combat 04: Shattered Skies | Action (77)           |
| Army Men 3D                    | FPS (77)              |
| Assassin's Creed               | Sports (76)           |
|                                | Fantasy (75)          |
|                                | RPG (75)              |
|                                | Turn-based (74)       |
|                                | Action Adventure (70) |



# BQL Relevance Models

---

Can define a relevance model to determine which results are more relevant to your query

◆ `USING RELEVANCE MODEL <model_definition>`

## Example:

```
SELECT year, genre, rating, _score
WHERE genre in ('puzzle', 'sports')
USING RELEVANCE MODEL my_model (favorite_genre:'puzzle', favorite_years:[1999, 2000])
  DEFINED AS (String favorite_genre, IntOpenHashSet favorite_years)
  BEGIN
    float boost = 0.0;
    if (favorite_years.contains(year)) boost += 100;
    if (favorite_genre.equals(genre)) boost += 50;
    return rating + boost;
  END
ORDER BY RELEVANCE;
```

# Other useful BQL information

---

Can create a parameter *at search time* for your query

- ◆ GIVEN FACET PARAM (*facet-name, param-name, param-type, param-value*)
- ◆ e.g. SELECT gname, **Network** WHERE **Network** in (0, 1) **GIVEN FACET PARAM (Network, "gid", int, 19);**

Can retrieve the stored data file (JSON file)

- ◆ e.g. SELECT gname, \_srcdata WHERE gid = 19 **FETCHING STORED;**

Can view meta-information using **DESCRIBE**

- ◆ Index name is optional



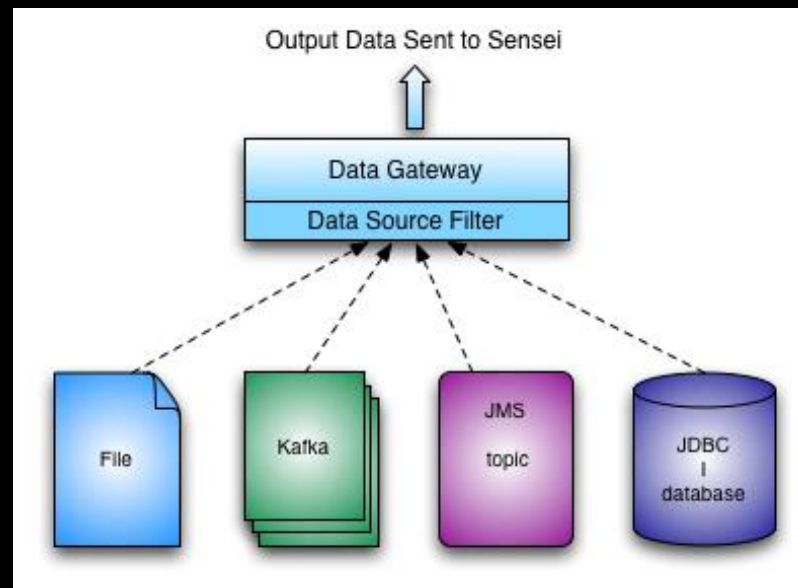
# Data population

Data is added (or removed) from Sensei with **data events**, which are units of indexing activity.

◆ Each data event is a tuple of (type, data, version).

Data events are streamed (consumed) through **Gateways**

- ◆ File
- ◆ JMS
- ◆ JDBC
- ◆ Kafka



# Data event examples

---

## Add data event example:

```
{"type":"add","data":
  {
    "id" : "1",
    "gname" : "Super Smash Bros. Brawl",
    "pubname" : "Nintendo",
    "releasedate" : "2008-03-09",
    "maxplayers" : "4",
    "rating" : "4.65",
    "genre" : "fighting,action,platformer",
    "description" : "Players choose from a large selection of characters and attempt
to knock there opponents off-screen as they fight"
  }
}
```

## Delete data event example:

```
{"type":"delete","id":1}
```



# *Interface with data in a program*

---

Several client libraries to interface with data

- ◆ Rest/JSON API over HTTP POST
- ◆ Java client API wrapping Rest API
- ◆ Python client API wrapping Rest API

Each API constructs a request object (represented as JSON) and sends the request to the Sensei node end-point

- ◆ Example end-point: <http://gpu1.ddl.ok.ubc.ca:50020/sensei>



# Hadoop/MapReduce Integration

---

Hadoop integration is supported

- ◆ Build an index using Hadoop first
- ◆ Retrieve the data using Sensei

MapReduce integration is technically possible, but...

“It's not that easy to extend the Sensei query functionality. At minimum you would need to implement your own FacetHandler, which would require **at least a week of ramping up with Bobo Architecture**. One will **need to understand how collectors, comparators, facets, scoring functions, explanations work**. **All these stuff is not easy to grasp**, especially because that code is performance critical and our team needed to consider trade-offs between readability and writing an extremely efficient code(reusing arrays, avoiding autoboxing, polymorphism, object creation, etc). Moreover you can not control how the results produced by the facet handler on the segment level, will be merged together on partition and cluster node level. That's why we didn't include group by by multiple columns and aggregation functions in the first release.”

# Challenges (B\*\*\*\*h'n)

---

Documentation is “okay” at its best...

- ◆ Lacks useful examples

Do not have any real examples for some features

- ◆ i.e. TIME predicates

Poor English, poor grammar... just poorly written

- ◆ Korean site was more helpful?

Errors in examples are atrocious!

- ◆ Spelling errors, inconsistent naming

Sensei is weird

- ◆ Does NOT support negative numbers!

## *In Our (Not So) Humble Opinion*

---

**DON'T USE SENSEI  
UNLESS YOU ARE  
WORKING FOR  
*LinkedIn!***

# References & Resources

---

- ◆ <http://senseidb.com/>
- ◆ <https://linkedin.jira.com/wiki/display/SENSEI/Home>
- ◆ <https://groups.google.com/forum/?fromgroups#!forum/sensei-search>
- ◆ <http://koikebox.tistory.com/category/%5B%EA%B2%80%EC%83%89%EC%97%94%EC%A7%84%5D/senseiDB>
- ◆ (South) Korea
- ◆ Ramon

