NoSQL DBMS based on a new Data Model

Ikarus DataBase Engine

Content:

Richard Hohensinner

1 Introduction:

A DataBase Management System requires a defined Data Model. This Data Model consists of two major pieces. On the one hand Data Objects/Structures have to be classified, on the other hand a set of Operations have to be defined. This leads us to the following scheme.

Data Model:

Data Model = <Data Objects, Operations>

Data Objects:

Data Object = <JSON file + unique ID>
<SCollection + unique ID>

Operations:

operation name(parameter1, parameter2,) => return_value

The NoSQL DBMS will build up on the database engine, implemented as a web service, as its foundation. It is using persistent data objects encoded as JSON files. These data objects are identifiable by their unique ID and can be combined into S-Collections which themselves are also identifiable by their own unique CID.

For the web service the Java API for creating XML web services, JAX-WS will be used (available with Java EE 1.6+).

The Database Engine will support (at least) the following operations:

- Store, Modify and Delete data objects
- Create and Delete S-Collections.
- Insert and Remove members into/from S-collections.
- Search data objects and S-collections;
- Scan a list of data objects and get the JSON.

The engine is reachable under "http://coronet2.iicm.tugraz.at:8080/IkarusDBEngine/"
A test client is reachable under "http://coronet2.iicm.tugraz.at:8080/IkarusDBEngineClient/"

2 Operations:

There are three core web service operations regarding JSON objects: **store**, **get**, and **delete**.
store creates a JSON object within the engine and automatically assigns a 6 digit ID to it.
get retrieves a JSON object, referenced by a 6 digit ID, previously stored within the engine.
delete erases a JSON object, referenced by a 6 digit ID, that is stored within the engine.

Five additional operations expand the engine's functionality by creating/manipulating/deleting sets of JSON objects, so called S-Collections(see X): **makecoll**, **deletecoll**, **insertcoll**, **removecoll** and **getcoll**.
makecoll creates a S-Collection by receiving a JSON object ID and a name for it. The object ID is automatically assigned as the head object of the recently created S-Collection.
deletecoll erases every information about the S-Collection, referenced by a 6 digit CID (with an additional pre-literal "s-") that is stored within the engine.
insertcoll adds a JSON object to an already existing S-Collection. The number of S-Collection members is limited by the maximum number of JSON objects existing at the same time (999.999).
removecoll eliminates a JSON object from a specific S-Collection. Note that the head object of a S-Collection cannot be removed, as S-Collections per definition require at least one member to exist.
getcoll behaves similar to get, but instead of returning the content of a JSON object it returns a String containing every member of an S-Collection (e.g. head_id1,id2,id3).

All operations require at least one String as an input and also return a String as an output. The reason behind this is, to keep the engine lightweight, easy to use and consistent. The operations regarding S-Collections only manipulate references to JSON objects, not the objects themselves. As a result, the manipulation of S-Collections may never change, corrupt or erase any data stored within the engine.
Two additional operations, a hard reset of the engine and a search feature, also belong to the basic set of operations . **searchobj** implements the basic functionality of a search filter. By entering a search text, all available data was browsed, resulting in a list with all ID's containing the search text. **reset** resets the web service/engine to a point, that is according to the first start of the engine. This operation is designed to only be used by administrators and thus should never be presented to the user directly.

2.1 JSON Objects:

2.1.1 store

store(String json_content) => String json_id

The store operation takes a JSON file, parsed as a String, as an input and will return the unique ID of the JSON file to be stored within the DataBase. The ID's will be assigned automatically during a successful invocation of the store operation. ID's will always consist of a 6 digit number - ranging from 000001 - 999999 as the last valid object ID. A call of store with an empty String (= null) will fail.

e.g.:

```
store("{example content...}")        => 000001
store("")                            => null
```

2.1.2 get

get(String json_id) => String json_content

The get operation takes a unique ID as an input. The ID passed to this operation has to follow the requirement of a 6 digit number, corresponding to the return value of store. Meaning that, 000001 would be accepted, but neither 1 not 001 would comply. The return value, in case of a call with a valid ID that is already stored within the DataBase, will return the JSON object parsed as a String. In case of an invalid call, either consisting of an invalid ID, or in case that nothing is stored within an maybe not yet existing object, referenced by the given ID, will fail.

e.g.:

```
get("000001")                        => "{example content...}")
get("01")                            => null  // invalid ID
get("")                              => null  // empty ID
get("012345")                        => null  // nothing stored
```

2.1.3 delete

delete(String json_id) => String json_id + " deleted"

The delete operation erases all information a specific JSON object stored within the DataBase, referenced by the given ID. Despite that, delete behaves very similar to the get operation and returns either, in case of a successful call the specified ID followed by a "deleted" String literal, or will fail, in case of an invalid call, e.g. the JSON file is not stored within the DataBase.

e.g.:

```
delete("000001")        => "000001 deleted"
delete("01")            => null  // invalid ID
delete("")              => null  // empty ID
delete("012345")        => null  // nothing stored
```

2.2 S-Collection Objects:

2.2.1 makecoll

makecoll(String coll_name, String head_id)
=> String coll_id + "(" + String coll_name + ")"

The makecoll operation takes two parameters as an input: The first parameter is a freely choose able name, used to give a human readable identifier besides the CID(Collection ID). The second parameter is the ID of the JSON object to be marked as HEAD for the new S-Collection. The makecoll operation will return a unique CID, automatically assigned at runtime, for the newly created S-Collection. The collection ID will consist of a String literal "s-" plus a 6 digit number (same requirements as needed for the JSON object ID) followed by the specified name in brackets.

e.g.:

```
makecoll("mycollection", "000001")   => "s-000001(mycollection)"
makecoll ("", "000001")              => null  // invalid name
makecoll ("test", "015")             => null  // invalid id
```

2.2.2 deletecoll

deletecoll(String coll_id)
=> String coll_id + "(" + String coll_name + ")" + " deleted"

The deletecoll operation takes the unique CID as an input. The return value is the same from the makecoll operation with an additional " deleted" String literal appended. Its additional behaviour is identical to the delete(json_id) operation.

e.g.:

```
deletecoll("s-000001")          => "s-000001(mycollection) deleted"
deletecoll ("000001")           => null // invalid coll_id
```

2.2.3 insertcoll

insertcoll(String coll_id, String json_id)
=> String json_id " successfully inserted into " String coll_id + "(" + String coll_name + ")"

The insertcoll operation takes 2 parameters as an input: Firstly the CID of the S-Collection to be inserted into and secondly the ID of the JSON object to insert. It will either succeed, given that the ID's are correctly entered in addition to the S-Collection already existing. It will fail, if any of the above mentioned requirements aren't fulfilled.

e.g.:

```
insertcoll("s-000001", "000002")
=> "000002 successfully inserted into s-000001(mycollection)"
insertcoll("000001" "000002")          => null  // invalid coll_id
insertcoll("s-000001", "002")          => null  // invalid json_id
```

2.2.4 removecoll

removecoll(String coll_id, String json_id)
=> String json_id " successfully removed from " String coll_id + "(" + String coll_name + ")"

The removecoll operation takes 2 parameters as an input: Firstly the ID of the S-Collection to be inserted into and secondly the ID of the JSON object to remove. It will either succeed, given that the ID's and the name are correctly entered in addition to the S-Collection already existing. It will fail, if any of the above mentioned requirements aren't fulfilled. Additionally, the removecoll operation will fail if the size of the S-Collection equals 1 (meaning that only the head object of the collection remains) - Existing S-Collections always require at least one element (= head), thus the head object can never be removed.

e.g.:

```
removecoll("s-000001, "000002")
=> "000002 successfully removed from s-000001(mycollection)"
removecoll("000001", "000002")                    => null  // invalid coll_id
removecoll("s-000001", "002")                     => null  // invalid json_id
```

2.2.5 getcoll

getcoll(String coll_id) => String head_and_members

The getcoll operation takes the ID of the S-Collection to be searched as an input. The return value is a String starting with the head of the S-Collection, followed by the other members of the collections - the elements are separated by commas ','. It will fail, if the S-Collection doesn't exist or the CID is invalid.

e.g.:

```
getcoll("s-000001")                               => "000001,000002,000004"
getcoll("000001")                                 => null  // invalid coll_id
getcoll("s-000420")                               => null // S-Coll. doesn't exist
```

2.3 Extra operations:

2.3.1 searchobj

searchobj(String search_text) => String all_ids_containing_text

The searchobj operation is a feature to browse the DataBase for a specific text or keyword. The user is able to enter a search text as a String, which can be of any size except null. As a result the searchobj operation returns a list with all the ID's containing the search text or keyword (same format as the getcoll operation). Note that it is also possible to enter an entire JSON object as a search text to get the specific ID of the JSON object.

e.g.:

```
searchobj("WhatDoesTheFoxSay")          => "00001, 000031"
searchobj("ThisIsNotPresentInData")     => null  // no id contains the text
searchobj("")                           => null  // empty passphrase
```

2.3.2 reset

reset(String passphrase) => String success

The reset operation is a feature to clear the DataBase without restarting the server completely. As a passphrase enter "IKnowWhatIamDoing" to clear all JSON objects and S-Collections stored within the DataBase as well as the automatically assigning ID counters.

e.g.:

```
reset("IKnowWhatIamDoing")
=> " Database was successfully cleared!"
reset("")                               => null  // empty passphrase
reset("IDontKnowWhatIamDoing")          => null  // wrong passphrase
```

3 Ikarus DataBase Engine:

3.1 Rationale

The reason Ikarus was created was bachelor thesis embedded in an international Project (NoSQL DBMS based on a new Data Model) involving the University of Technology (Graz - Austria) as well as the Saint Petersburg University of Technology & Design.
As a consequence, Ikarus was implemented as a web service, using Java as a programming language, in order to make it as versatile and accessible as possible. JAX-WS, the Java API for creating XML based Web Services, was used as its core component. JAX-WS is available with Java EE 1.6+. The client and server communicate through SOAP messages, which makes it independent from protocols and transportation. JAXB, as a delegate of JAX-WS translates XML into Java conforming data objects. This makes using JAX-WS as convenient and lightweight as possible.

3.2 Source Code

Two main source files have to be mentioned as they form the core functionality of the engine, CoreWS.java and Main.java. Besides those two files, small Utility files (e.g. implementing S-Collections) replenish Ikarus.

3.2.1 CoreWS.java

To start off with the basics, for every web service operation, a web method was created. These web methods/operations are combined in a single java file called CoreWS.java:

Figure 1)

Figure 2)

Figure 1) and 2) show a graphical illustration of the 10 web service operations with names and input parameters.

To give an example of what a web method looks like as code:

```
/**
* Web service operation
*/
@WebMethod(operationName = "store")
public String store(@WebParam(name = "content") String content)
{
   if(content == null)
   {
      return "content NULL";
   }

   String new_id = Main.getInstance().getNewJsonId();
   Main.getInstance().addNewJson(new_id, content);

   // server log:
   System.out.println("STORE: " + new_id + " " + content);

   return new_id;
}
```

Besides checking the input parameters and returning the output string nothing special happens in the web service operation class directly. Its main functionality is outsourced to another file, Main.java.

3.2.2 Main.java

The Main.java class forms the heart piece of Ikarus as it acts as a manager behind the scenes. Its main attribute is defined in being a static instance of class, granting singleton like behaviour - simply put, there is exactly one instance of the main class that is omnipresent, visible to all operations and cannot be destroyed.

As mentioned above, the Main class implements the web service operations in detail. Besides taking care of storing, manipulating and freeing data by handling data structures, Main.java also ensures accurate behaviour of the web methods.

For storing available data, three maps are used:

```
// id (e.g. "000001") - content (e.g. "{yljasldamy.xc}")
public static Map<String, String> json_map = new HashMap<String, String>();
// id (e.g. "S-000001") - name (e.g. "examplecollection")
public static Map<String, String> coll_name_map = new HashMap<String, String>();
// id (e.g. "S-000001") - s-coll-objects (e.g. "examplecollection")
public static Map<String, SCollection> s_coll_map = new HashMap<String, SCollection>();
```

To conclude, the Main class is the most important component of Ikarus, not only does it ensure uniqueness of the engine and omnipresent behaviour through its instance-like implementation, but also avoid race conditions by having a static attribute, realizing a task-queueish purpose.

3.3 Execution

Ikarus is currently online at:

http://coronet2.iicm.tugraz.at:8080/IkarusDBEngine/

In order to invoke Ikarus' web service operations, SOAP messages have to be sent to the running engine. As this is not a trivial task, Ikarus also includes a basic test client extension that was created to illustrate Ikarus' correct behaviour.

4 Ikarus DataBase Engine Client:

TODO

5 Additional Information:

5.1 JSON

5.2 S-Collections