NoSQL DBMS based on a new Data Model

1.) Ikarus DataBase Engine:

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 Objects, Operations>

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 ID.

For the web service the Java API for creating XML web services, JAX-WS will be used (available with Java EE 6+).

// Because Prof. Scerbakov was kind enough to provide us with a real Tomcat server, final
// testing and deployment will be executed there. (http://coronet2.iicm.tugraz.at/...)

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.

Data Objects:

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

Operations:

- JSON Objects:

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 stored JSON file 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) 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, similar to the return value of STORE - 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 String; In case of an invalid call, either consisting of an invalid ID or the fact that nothing is stored pointed to, 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

3) DELETE

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

The DELETE operation behaves very similar to the GET operation and return either, in case of a successful call the specified ID followed by a "deleted" text, 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

- S-Collection Objects:

4) MAKECOLL

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

The MAKE COLL(ection) 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 coll_id. The second parameter is the ID of the JSON file to be marked as HEAD for the new S-Collection. The MAKECOLL operation will return a unique coll_id 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

5) DELETECOLL

DELETECOLL(String coll_id, String coll_name)
=> String coll_id + "(" + String coll_name + ")" + " deleted"

The DELETE COLL(ection) operation takes the unique collection ID and its corresponding name as an input. The return value is the same from the MAKECOLL operation with an additional " deleted" message appended. Its additional behaviour is identical to the DELETE(json object) operation.

e.g.: DELETECOLL(s-000001, "mycollection") => "s-000001(mycollection) deleted"
MAKECOLL (" ", s-000001) => null // invalid name
MAKECOLL ("test", 000001) => null // invalid coll_id

6) INSERTCOLL

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

The INSERT COLL(ection) operation takes 3 parameters as an input: Firstly the ID of the S-Collection to be inserted into, secondly the name of the S-Collection and thirdly the ID of the JSON object to insert. 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.

e.g.: INSERTCOLL("s-000001", "mycollection", "000002")
=> "000002 successfully inserted into s-000001(mycollection)"
INSERTCOLL("000001" ...) => null // invalid cid
INSERTCOLL(... " " ...) => null // empty name
INSERTCOLL(... "001") => null // invalid id

7) REMOVECOLL

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

The REMOVE COLL(ection) operation takes 3 parameters as an input: Firstly the ID of the S-Collection to be inserted into, secondly the name of the S-Collection and thirdly 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", "mycollection", "000002")
=> "000002 successfully removed from s-000001(mycollection)"
REMOVECOLL("000001" ...) => null // invalid cid
REMOVECOLL(... " " ...) => null // empty name
REMOVECOLL(... "001") => null // invalid id

8) GETCOLL

GETCOLL(String coll_id, String coll_name) => String head_and_members

The GET COLL(ection) operation takes 2 parameters as an input: First the ID of the S-Collection to be searched, second the collection name of the (already existing) S-Collection, linked to the ID. 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 parameters are invalid.

e.g.: GETCOLL(s-000001, "mycollection") => "000001,000002,000004"
GETCOLL("000001" ...) => null // invalid cid
GETCOLL(... " " ...) => null // empty name
GETCOLL("s-000007", "notexistingcoll") => null // S-Coll not existing

8) 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