

Globals: A Universal NoSQL Engine

Rob Tweed (rtweed@mgateway.com web: <http://www.mgateway.com>)

Acknowledgement

This paper is adapted from “A Universal NoSQL Engine, Using a Tried and Tested Technology” by Rob Tweed and George James (2010) (<http://www.mgateway.com/docs/universalNoSQL.pdf>)

Introduction

Globals (<http://globalsdb.org>) is a free database that uses a type of data storage known as Global Persistent Variables. These are an abstraction of the underlying B-tree structures and can be used to store large volumes of data as multi-dimensional sparse arrays. Global Persistent Variables (usually simply referred to as "Globals" – hence the name of the database) are an expressive and highly efficient way of modelling all of the common use cases that are targeted these days by NoSQL databases.

The database engine at the heart of Globals is, in fact, the same one that is used at the core of the Caché database (<http://www.intersystems.com/cache/index.html>). The key benefit for the Globals database developer is that, unlike most of the major NoSQL databases, Globals is based on a mature, tried and tested database technology with an enviable pedigree of massive scalability and extremely high performance in business critical environments such as healthcare and financial services. Unlike Caché, Globals is free to use, deploy and distribute without any restrictions, making it a very attractive platform (note that it is not an Open Source product, however).

This paper explains how the Globals database can be used to model and store data from each of the distinct types of NoSQL databases: Key/value store, Tabular/Column store, Document store and Graph database, and how, as a result, you have the best of all worlds: NoSQL capabilities combined with the performance, reliability and maturity needed for real-world business-critical applications.

Globals can be accessed via very high-performance in-process APIs that are available for Java and Node.js (Javascript). Access via other languages is apparently planned in future. This paper will focus on the Node.js Javascript interface.

<i>Note: Globals are not to be confused with the more commonly used term that refers to globally scoped variables. In GlobalsDB, Globals are a data storage structure.</i>
--

A Brief Introduction to NoSQL

The term NoSQL has been around for just a few years and was invented to provide a descriptor for a variety of database technologies that emerged to cater for what is known as "Web-scale" or "Internet-scale" demands.

Put simply, there are three aspects to web-scale:

- big data: the biggest of the web applications out there (eg Twitter, Facebook, Google etc) are handling quantities of data that are orders of magnitude greater than anything previously considered for database management
- huge numbers of users: numbered in the millions, accessing systems concurrently and constantly
- complex data: typically these applications aren't handling the simple tabular data that one finds in many commercial and business applications.

The relational database technologies that have dominated the IT industry since the 1980s began to show their weaknesses in these three areas when pushed to Web-scale, so a growing number of people began looking for alternatives. And so the NoSQL databases began to emerge. Although a variety of models have evolved, they all tend to follow similar patterns:

- they handle the huge quantities of data by breaking it up across servers, a process known as sharding
- they handle the huge numbers of users by spreading the load across servers: ie by using parallel processing
- they use simpler, more flexible schema-free database designs

Without exception, the most successful and well-known of the NoSQL databases have been developed from scratch, all within just the last few years. Strangely, it seems that nobody looked around to see whether there were any existing, successfully-implemented database technologies that could have provided a sound foundation for meeting Web-scale demands. As this paper aims to demonstrate, Globals turns out to be a perfect basis for meeting the objectives of NoSQL databases, using an existing, tried and tested database technology. Combined with the extremely high performance of Node.js, the Globals database provides an ideal, industrial-strength, high-performance platform for implementing NoSQL capabilities.

A Quick Overview of Globals

To avoid confusion, from this point on in this paper, we'll be referring to the database product as GlobalsDB, and when we refer to Globals, we'll be referring to the data storage technology that underpins the GlobalsDB product.

Globals are:

- schema-free
- hierarchically structured
- sparse
- dynamic

Think of a persistent associative array and you're on the right kind of track. Some examples of Globals would be:

```
myTable["101-22-2238", "Chicago", 2] == "Some information"  
account["New York", "026002561", 35120218433001] == 123456.45
```

Each Global has a name (*cf* array name). There then follows a number of subscripts whose values can be numeric or text strings. You can have any number of subscripts. Each Global "node" (a node is defined by a Global name and a specific set of subscripts) stores a data value which is a text string (empty strings are allowed).

You can create or destroy global nodes whenever you like. They are entirely dynamic and require no pre-declaration or schema.

It is up to you, the developer, to design the higher-level abstraction and meaning of a database that is physically stored as a set of Globals. Globals provide no built-in indexing, so to provide the performance needed for searching and querying your data, you must create and maintain additional Global nodes that represent indices.

Globals turn out to be extremely versatile, and can be very easily used to model all four types of NoSQL database with comparable or, in many cases, significantly higher, levels of performance:

- key/value stores (eg Redis, memcached)
- tabular, column-orientated databases (eg BigTable, Cassandra, SimpleDB)
- document databases (eg CouchDB, MongoDB)
- graph databases (eg Neo4j)

We'll see later some detailed examples of how this can be done.

In-memory Performance, On-disk Integrity

One of the key reasons for the extremely high performance of GlobalsDB is the sophisticated caching mechanisms that have been refined and optimised over the years in Caché, from which GlobalsDB inherits its core database engine. This caching ensures that, for most of the time, the Global nodes to which you require access are already in memory. Most NoSQL databases still have an immature relationship between in-memory activity (for speed) and on-disk activity (for persistence and integrity) and are open to risk if a node or shard server goes down (*eg* <http://highscalability.com/blog/2010/10/15/troubles-with-sharding-what-can-we-learn-from-the-foursquare.html>). By comparison, Caché, and as a result, GlobalsDB has been hardened against such risks. The result is tried and tested in-memory-like performance with on-disk-like integrity, the direct result of decades of real-world experience in demanding environments such as healthcare, finance and banking, where performance, integrity and reliable non-stop operation are mission-critical.

It turns out the goals that have driven and defined the development of the database engine inside GlobalsDB are the very same goals of the NoSQL movement.

Ticking the NoSQL Boxes

In 2010, TechRepublic published a blog named "10 things you should know about NoSQL databases" (<http://blogs.techrepublic.com.com/10things/?p=1772>). This listed 5 advantages of and 5 challenges for NoSQL databases. In fact, if you judge GlobalsDB against these 10 criteria, it scores a tick against all 5 advantages, but, more interestingly, address 4 out of 5 of the challenges, something no other NoSQL database can boast. Let's quickly look at these criteria:

5 Advantages:

- Elastic Scaling
- Big Data
- Goodbye DBAs
- Economics
- Flexible Data Models

5 Challenges:

- Maturity
- Support
- Analytics and Business Intelligence
- Administration
- Expertise

Taking each advantage in turn:

- **Elastic scaling:** Caché has, for many years, supported scaling out across multiple servers, and can do so across low-cost commodity hardware. In the case of Caché, their ECP networking technology allows seamless logical views of Globals, where that global can be physically distributed across multiple servers. GlobalsDB does not actually include the ECP networking capabilities of Caché, but if such elastic scaling is required, a GlobalsDB application can be directly migrated to Caché: Caché will run any GlobalsDB application without any changes being necessary.
- **Big Data:** GlobalsDB is designed to support and manage huge volumes of data, way beyond the limits of relational databases, whilst still delivering extremely high performance.
- **Goodbye DBAs:** interestingly, InterSystems, the vendor of Caché, has used this feature in its marketing for many years. There are stories of Global-based systems that have been running unattended for decades.
- **Economics:** GlobalsDB will happily run on low-cost, commodity hardware and extract maximum levels of performance from them. Partners Healthcare in Massachusetts supported tens of thousands of interactive users throughout

the 1980s and 1990s on a networked cluster of hundreds of commodity MSDOS-based PCs running a pre-cursor to Caché.

- **Flexible Data Models:** this is the very essence of Global storage as this paper will later explore.

As to the challenges:

- **Maturity:** Unlike the new NoSQL databases, Caché has a long pedigree and outstanding track record, supporting large complex databases in demanding, real-world business environments. GlobalDB, which shares its database engine, is, as a result, a robust, extremely reliable and stable technology that can be confidently used in business-critical situations
- **Support:** Caché dominates the healthcare industry and is also used extensively in the financial services industry. One of the key reasons that it is entrusted to such business- and safety-critical roles has been the quality of the commercial support that is provided by InterSystems. GlobalDB is provided free of charge but without any support, but if support is required, a GlobalDB application can be migrated without any changes needed to the fully supported Caché product.
- **Analytics and Business Intelligence:** interestingly, InterSystems are now heavily marketing a product called DeepSee, which is designed for exactly this purpose, and is layered on top of Caché. Caché supports connectivity to SQL-based business analytics tools: for many years they have supported both SQL and non-SQL database access. So, if such functionality is required, a GlobalDB application can be migrated, without change, to Caché.
- **Administration:** GlobalDB is straightforward to install and maintain, and, like Caché, can be used in situations where there are few, if any, skilled IT resources available.
- **Expertise:** This is the one area where GlobalDB admittedly fall short. Whilst the user-base for Caché has been steadily growing, the number of skilled professionals who have experience in using global-based databases is very small compared with the availability of RDBMS skills. However, one of InterSystems' states objectives for GlobalDB is as a way of reaching a new community of developers and users who otherwise might not consider Caché.

The free GlobalDB database is, in summary, an ideal candidate for businesses that require a NoSQL technology but require something that is mature, robust and reliable. An easy and ready migration path to the commercial Caché database provides optional access to additional high-end NoSQL functionality and capabilities.

Modelling the 4 NoSQL Database Types Using GlobalsDB

Before we describe how each NoSQL database type can be modelled in GlobalsDB, let's first examine the structure of Globals (ie the data storage structures) in a little more detail and define some of the terminology that we'll use later.

When storing data in Globals, there are three components that you use to represent a unit of storage:

- Global name.
- subscripts (zero, one or more). These may be text strings or numeric values
- value (the value to be stored). This may be a text string or a numeric value

They are commonly expressed as [n]-ary relational variables in the following form:

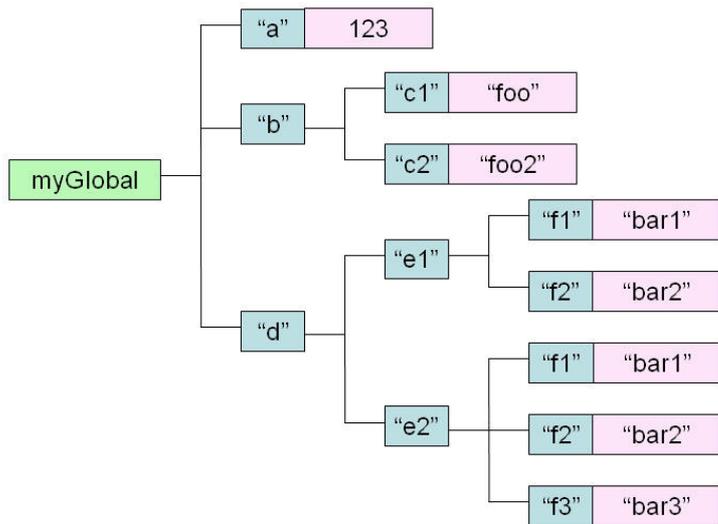
```
globalName[subscript1, subscript2, ..subscriptn] == value
```

This combination of name, subscripts and value is known as a *Global Node* and is the unit of storage. A Global is a collection of Global Nodes and a database is a collection of Globals.

An important aspect of Globals is that a single named Global can contain nodes with different numbers of subscripts, eg:

```
myGlobal["a"] == 123
myGlobal["b", "c1"] == "foo"
myGlobal["b", "c2"] == "foo2"
myGlobal["d", "e1", "f1"] == "bar1"
myGlobal["d", "e1", "f2"] == "bar2"
myGlobal["d", "e2", "f1"] == "bar1"
myGlobal["d", "e2", "f2"] == "bar2"
myGlobal["d", "e2", "f3"] == "bar3"
```

The net result is that a single named Global represents a sparse hierarchical tree of nodes. For example, the Global above effectively represents the following hierarchical tree:



You can create as many different named Globals as you like. So in other words, a database in GlobalsDB will consist of one or more named Globals, each with its hierarchy of nodes.

There is no explicit relationship between the named Globals in such a database, but there may be implicit relationships that are determined and managed at the application level. There is no explicit schema associated with Globals, and the way in which data is represented within Global nodes is implicitly defined at the application level.

Global nodes are created by using the *set* command. The exact syntax for this command depends on the API you use. So, for example, if you were using the Node.js API, you would create this Global Node in one of two ways:

Asynchronously:

```

var globalNode = {
  global: "myGlobal",
  subscripts: ["b", "c3"],
  data: "Rob"
};
db.set(globalNode, function(error, results) {
  // etc
});

```

or Synchronously:

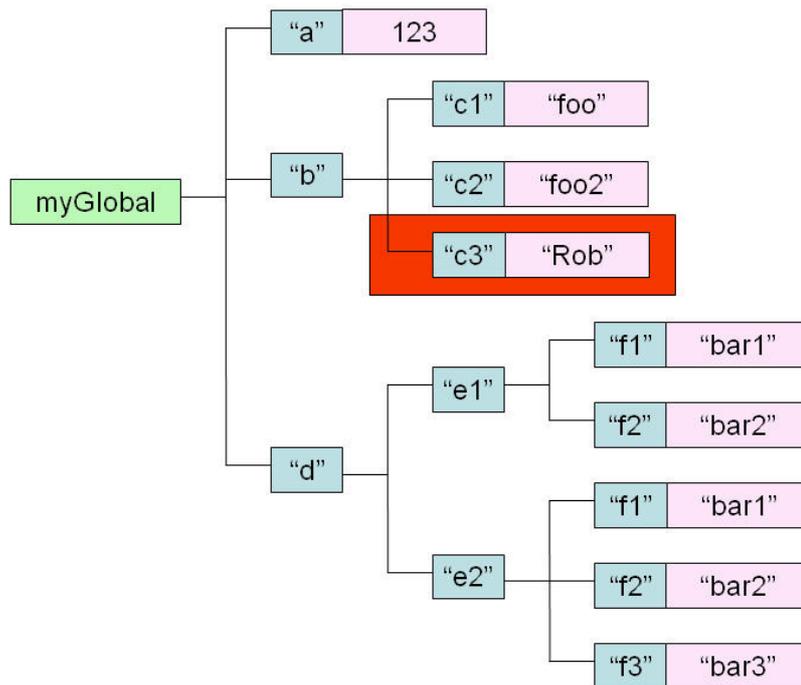
```

var globalNode = {
  global: "myGlobal",
  subscripts: ["b", "c3"],
  data: "Rob"
};
db.set(globalNode);

```

One of the exciting things about GlobalsDB is that because it runs in-process via its Node.js interface, and because of its in-memory performance, unlike any other database (including the most popular of the NoSQL databases), you can afford to use synchronous coding, making it significantly easier to work with and allowing the full use of OO syntax within Javascript.

Invoking this command would insert this node to our hierarchy and the tree would now look like the following:



So, with this basic information in mind, now let's look at how you can use Globals for representing the data structures typically found in NoSQL databases.

1) Key/Value Storage

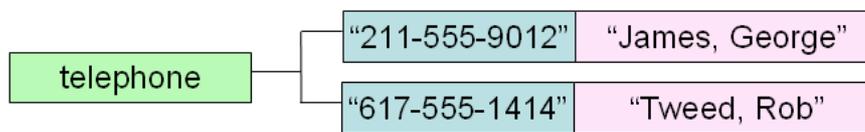
Implementing a key/value store using Globals is simple. You could create a very basic one using the following Global structure:

```
keyValueStore[key] == value
```

eg:

```
telephone["211-555-9012"] == "James, George"  
telephone["617-555-1414"] == "Tweed, Rob"
```

Viewed as a hierarchical tree:



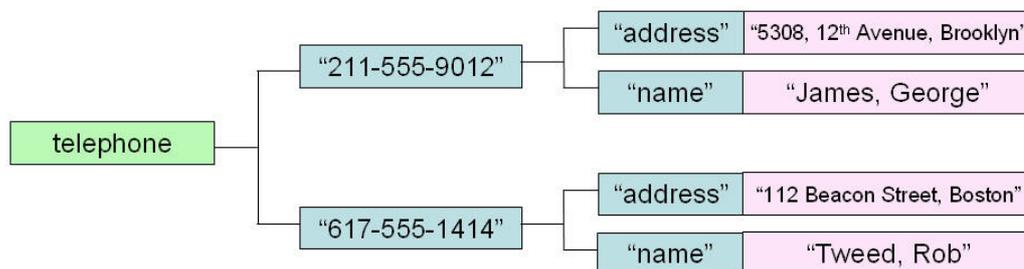
That's all there is to implementing simple key/values stores, but with Globals you can go further by storing multiple attributes against each key. For example:

```
Telephone[phoneNumber, "name"] == value  
Telephone[phoneNumber, "address"] == value
```

eg:

```
telephone["211-555-9012", "name"] == "James, George"  
telephone["211-555-9012", "address"] == "5308, 12th Avenue, Brooklyn"  
  
telephone["617-555-1414", "name"] == "Tweed, Rob"  
telephone["617-555-1414", "address"] == "112 Beacon Street, Boston"
```

ie we've now created a hierarchical tree that looks like this:



For example, to create the first record in this redesigned key/value store using the Node.js API:

```

var gnode = {
  global:'telephone',
  subscribers: ['617-555-1414','name'],
  data: 'Tweed, Rob'
};
db.set(gnode);

```

NoSQL databases typically don't provide automatic methods for indexing data. Neither do Globals. If you want to access data via an alternative key then you simply create a second Global with the alternate item as a key.

For example if you wanted to access the data using name as a key you'd need to add an index Global and update it at the same time as the telephone Global. Designing and adding indices is entirely your responsibility, but is very simple.

Here we add a simple name index by creating the following global nodes every time we add entries to the telephone Global:

```

nameIndex[name, phoneNumber] == ""

```

eg:

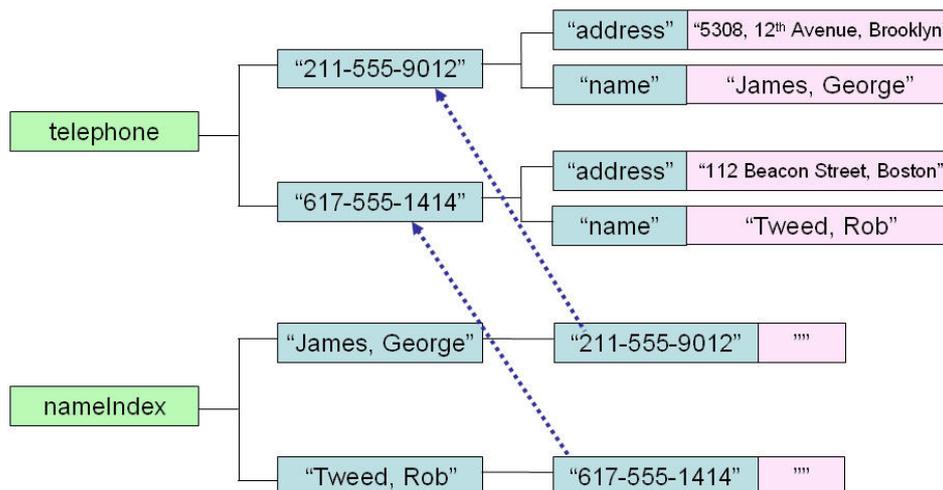
```

nameIndex["James, George", "211-555-9012"] == ""
nameIndex["Tweed, Rob", "617-555-1414"] == ""

```

No data value is necessary for this index, so we just use an empty string.

Viewing the main data Global and our index Global together diagrammatically (the dotted lines show the implied relationships between the index and telephone data nodes):



This index global provides us with a method of accessing our telephone data by name, while the main global allows us to access the data by telephone number.

A very important and powerful feature is that Global Nodes are automatically stored as a sorted set within a Global (as shown in the diagram above). An iterator method is provided that enables the content of every Global to be accessed sequentially. If we wanted to produce a telephone directory from this data we can iterate through the *nameIndex* Global and then use a *get()* method to access the address data from the telephone Global.

The iterator is known as the **order** function. For example using the the Node.js API:

```
gnode = {
  global:'nameIndex',
  subscrip:['James, George']
};
var nextName = db.order(gnode).result;
```

would return the next Global Node subscript following 'James, George' in this Global *ie*:

```
nextName == 'Tweed, Rob'
```

GlobalsDB is very highly optimised for traversing subscripts in this way, so if you design your indices well, searching for data in Globals is exceptionally fast.

There are all kinds of ways in which Globals can be designed for use as simple key/value stores. For example, you could redesign our example store using just one single global for both data and indices by adding a further first subscript, eg:

```
telephone["data", phoneNumber, "address"] == address
telephone["data", phoneNumber, "name"] == name
telephone["nameIndex", name, phoneNumber] == ""
```

Since the physical implementation of Globals is abstracted away, you can design the structure of your Globals to precisely match your processing needs. However, if your key/value stores are going to grow to enormous sizes, then you'll need to consider if and how a particular structure will aid or hinder the management of the Global(s) concerned (eg in terms of backup and recovery, maximum database size limits, distribution across multiple shards etc). This kind of consideration can affect whether you store data in one single named Global or spread it across several differently-named Globals.

Other Key/Value Types

If you look at a NoSQL key/value store such as Redis, you'll find that it provides a variety of other types. It turns out that every one of these types can also be very simply implemented using Globals.

Lists

Redis List types are linked lists. You can push values onto a list and pop values off a list, return a range of values, etc.

To model such a structure using Globals is pretty straightforward. For example you could use a structure such as the following:

```
list[listName, "firstNode"] == nodeNo
list[listName, "lastNode"] == nodeNo
list[listName, "node", nodeNo, "value"] == value
list[listName, "node", nodeNo, "nextNode"] == nextNodeNo
list[listName, "node", nodeNo, "previousNode"] == prevNodeNo
```

eg a linked list named *myList* that contains the sequence of values:

- Rob
- George
- John

could be represented as:

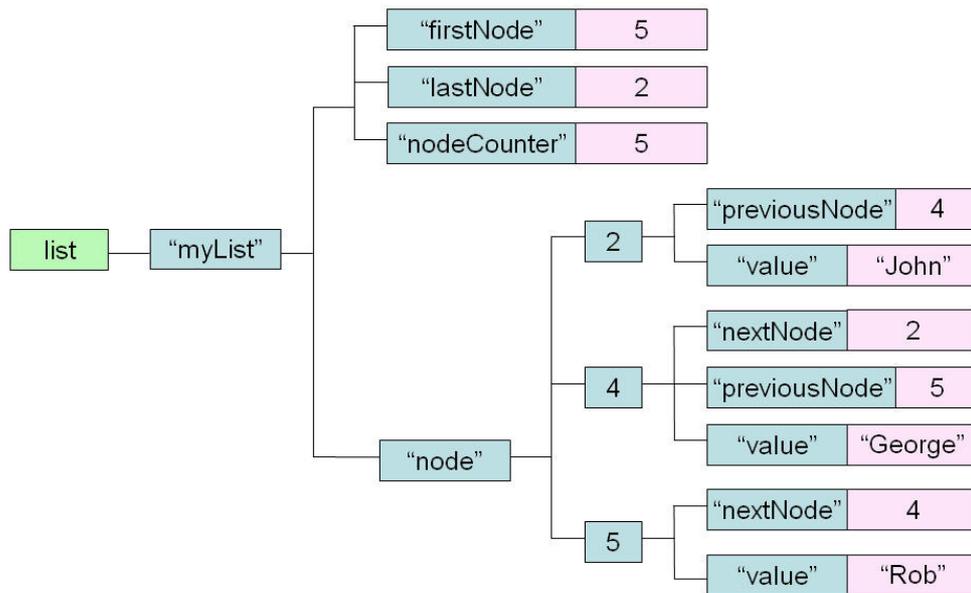
```
list["myList", "firstNode"] == 5
list["myList", "lastNode"] == 2
list["myList", "nodeCounter"] == 5

list["myList", "node", 2, "previousNode"] == 4
list["myList", "node", 2, "value"] == "John"

list["myList", "node", 4, "nextNode"] == 2
list["myList", "node", 4, "previousNode"] == 5
list["myList", "node", 4, "value"] == "George"

list["myList", "node", 5, "nextNode"] == 4
list["myList", "node", 5, "value"] == "Rob"
```

or diagrammatically:



What you see here is the sparse structure of Globals. The node numbers are just sequential integer values. Node 5 is currently the first record in the list, so it has an attribute that indicates the next node, but it doesn't have an attribute to indicate a previous node as there isn't one. The middle node (#4) has attributes for both next and previous nodes.

Each operation that modifies the list, eg pop, push, trim etc, would need to modify a set of nodes within this structure, eg:

- Resetting the first or last node pointer
- Adding or deleting a new node value
- Resetting the relevant next and previous node pointers to add a new node into the list, or to remove a node from the list

So, for example, pushing a new name, "Chris", onto the top of the list would change the list Global as follows:

```

list["myList", "firstNode"] == 6
list["myList", "lastNode"] == 2
list["myList", "nodeCounter"] == 6

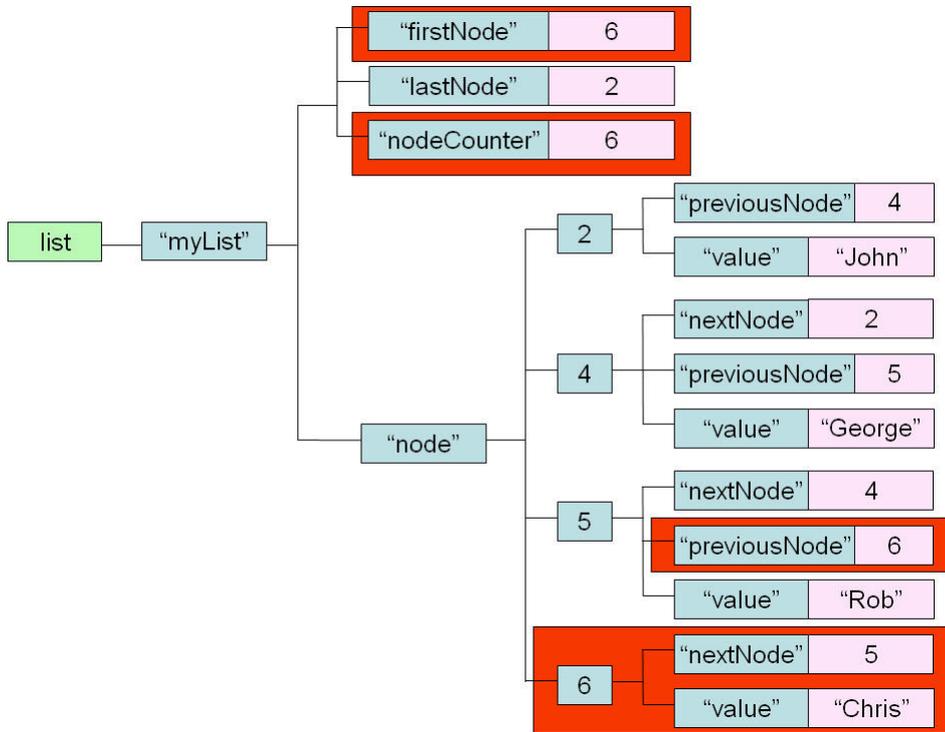
list["myList", "node", 2, "previousNode"] == 4
list["myList", "node", 2, "value"] == "John"

list["myList", "node", 4, "nextNode"] == 2
list["myList", "node", 4, "previousNode"] == 5
list["myList", "node", 4, "value"] == "George"

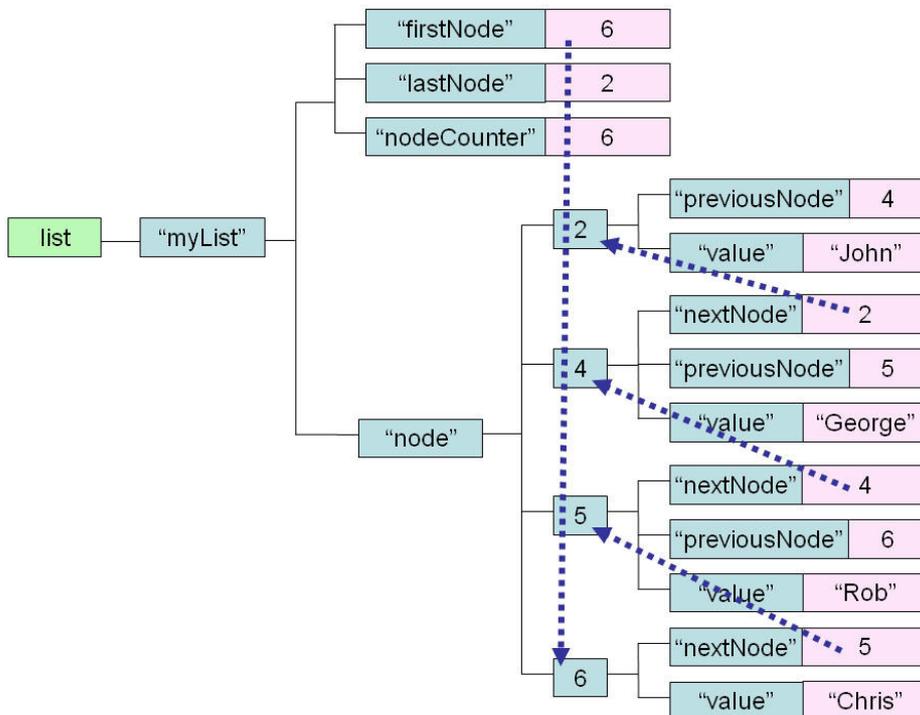
list["myList", "node", 5, "nextNode"] == 4
list["myList", "node", 5, "previousNode"] == 6
list["myList", "node", 5, "value"] == "Rob"

list["myList", "node", 6, "nextNode"] == 5
list["myList", "node", 6, "value"] == "Chris"
  
```

ie diagrammatically (changes to previous hierarchy are highlighted):



Traversing this list would involve starting at the first node and recursively following the nextNode records until no more are found, ie:



Returning a count of the number of records in the list could be done by traversal of the list, or, for maximum performance, it could be stored as a separate Global node and updated whenever the list is modified, eg:

```
List["myList", "count"] == noOfNodes
```

Clearly we'd need to implement these operations as methods that manipulate the Global nodes in this model appropriately, but this would be a very simple task.

Sets

Redis Sets are unordered collections of strings. We can easily model something with the same behaviour using Globals:

```
theSet[setName, elementValue] == ""
```

In fact, you'll notice that this is identical to how we defined an index earlier on. So we can add an element to a set:

```
Set: theSet["mySet", "Rob"] == ""
```

We can remove an element from a set:

```
Kill: theSet["mySet", "Rob"]
```

To determine if an element exists or not in a set, we make use of the **data** command. This will return 1 if the element exists and 0 if it doesn't:

```
Data: theSet["mySet", "Rob"] → 1  
Data: theSet["mySet", "Robxxx"] → 0
```

For example, if you use the Node.js API, you can use the **data** function

```
gnode = {  
  global:'theset',  
  subscripts:['mySet', 'Rob']  
};  
var exists = db.data(gnode).defined;
```

In the example above, exists will have a value of 1

We can use the natural ordering of Global Nodes to list the members of a set in alphanumeric sequence. By using Globals, there really isn't any substantial difference between Redis's **sets** and **zsets** when modelled using Globals.

Hashes

By now you can probably see that hashes can be implemented in exactly the same way as Sets. In fact, Globals are essentially persistence hash tables anyway.

```
Hash[hashName, value] == ""
```

2) Tabular (or Columnar) Storage

Table-based or Column-based NoSQL databases such as BigTable, Cassandra and Amazon SimpleDB allow data to be stored as sparse tables, meaning that each row in a table can have a value in some, but not necessarily all, columns. SimpleDB goes further and allows a cell in a column to contain more than one value.

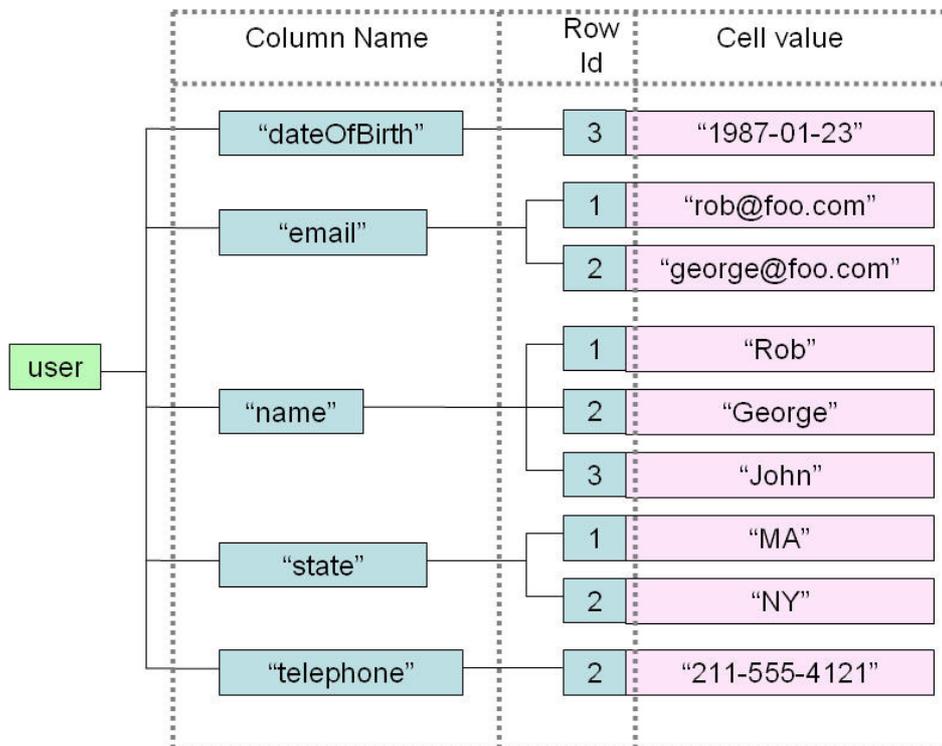
Once again, it turns out that Globals can be used to model such a data store. For example, the following structure would provide the basic features of such a store:

```
columnStore[columnName, rowId] == value
```

eg:

```
user["dateOfBirth", 3] == "1987-01-23"  
user["email", 1] == "rob@foo.com"  
user["email", 2] == "george@foo.com"  
user["name", 1] == "Rob"  
user["name", 2] == "George"  
user["name", 3] == "John"  
user["state", 1] == "MA"  
user["state", 2] == "NY"  
user["telephone", 2] == "211-555-4121"
```

Or diagrammatically:



Once again, the sparse nature of Globals comes to the fore. The Global above represents the following table:

	name	telephone	email	dateOfBirth	state
1	Rob		rob@foo.com		MA
2	George	211-555-4121	george@foo.com		NY
3	John			1987-01-23	

We could, of course, add indices to this model, eg by row and by cell value, that would be maintained in parallel with the main column store global, eg:

```
userIndex["byRow", rowId, columnName] == ""  
userIndex["byValue", value, columnName, rowId] == ""
```

3) "Document" Storage

Document-oriented NoSQL databases such as CouchDB and MongoDB store collections of key/value pairs and recursively collections of collections. Typically, JSON, or JSON-like structures, are used to represent these "documents".

GlobalsDB automatically maps JSON documents or objects to Globals.

For example, consider the JSON document:

```
{key: "value"}
```

This can be modelled as a Global as follows:

```
Document["key"] == "value"
```

In GlobalsDB we can create this document as follows:

```
var json = {
  node: {
    global: "Document",
    subscripts: []
  },
  object: {
    key: "value"
  }
};
db.update(json, 'object');
```

and retrieve it as follows:

```
var json = db.retrieve({global: 'Document'}, "object");
console.log("Document = " + JSON.stringify(json.object));
```

Let's take a more complex document:

```
{this:{looks:{very:"cool"}}}
```

could be represented as the following Global node:

```
Document["this", "looks", "very"] == "cool"
```

We could create this as follows:

```
var json = {
  node: {
    global: "Document",
    subscripts: []
  },
  object: {
    this: {
      looks: {
        very: 'cool'
      }
    }
  }
};
db.update(json, 'object');
```

How about the array:

```
["this", "is", "cool"]
```

This would be mapped to:

```
document[1] == "this"
document[2] == "is"
document[3] == "cool"
```

GlobalsDB actually maps to and from objects rather than arrays, so to create these nodes we'd do the following:

```
var json = {
  node: {
    global: "Document",
    subscripts: []
  },
  object: {
    1: "this",
    2: "is",
    3: "cool"
  }
};
db.update(json, 'object');
```

and retrieving it, we'd get:

```
var json = db.retrieve({global: 'Document'}, "object");
console.log("Document = " + JSON.stringify(json.object));
```

```
Document = {"1":"this","2":"is","3":"cool"}
```

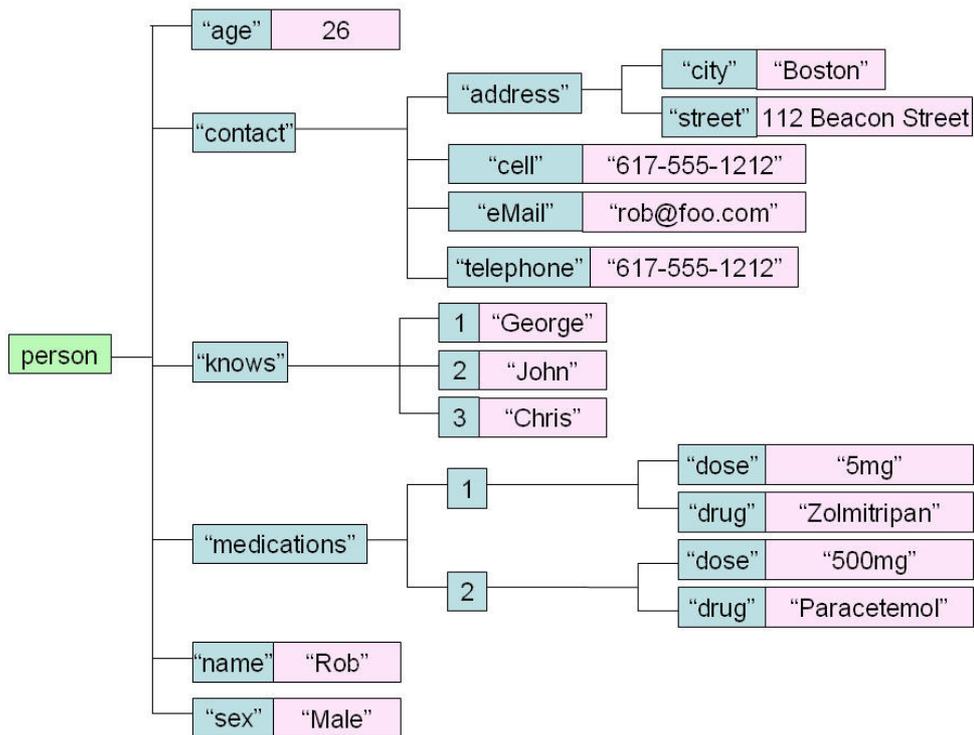
Put together a more complex JSON document:

```
{
  "age": "26",
  "contact": {
    "address": {
      "city": "Boston",
      "street": "112 Beacon Street"
    },
    "cell": "617-555-1761",
    "email": "rob@foo.com",
    "telephone": "617-555-1212"
  },
  "knows": {
    "1": "George",
    "2": "John",
    "3": "Chris"
  },
  "medications": {
    "1": {
      "dose": "5mg",
      "drug": "Zolmitripan"
    },
    "2": {
      "dose": "500mg",
      "drug": "Paracetamol"
    }
  },
  "name": "Rob",
  "sex": "Male"
}
```

And this would map to:

```
person["age"] == 26
person["contact", "address", "city"] == "Boston"
person["contact", "address", "street"] == "112 Beacon Street"
person["contact", "cell"] == "617-555-1761"
person["contact", "eMail"] == "rob@foo.com"
person["contact", "telephone"] == "617-555-1212"
person["knows", 1] == "George"
person["knows", 2] == "John"
person["knows", 3] == "Chris"
person["medications", 1, "drug"] == "Zolmitripan"
person["medications", 1, "dose"] == "5mg"
person["medications", 2, "drug"] == "Paracetamol"
person["medications", 2, "dose"] == "500mg"
person["name"] == "Rob"
person["sex"] == "Male"
```

Or diagrammatically:



We would create this document in GlobalsDB as follows:

```

var json = {
  node: {
    global: 'person',
    subscripts: []
  },
  object: {
    name: 'Rob',
    age: 26,
    knows: {
      1: 'George',
      2: 'John',
      3: 'Chris'
    },
    medications: {
      1: {
        drug: 'Zolmitripan',
        dose: '5mg'
      },
      2: {
        drug: 'Paracetamol',
        dose: '500mg'
      }
    },
    contact: {
      email: 'rob@foo.com',
      address: {
        street: '112 Beacon Street',
        city: 'Boston'
      },
      telephone: '617-555-1212',
      cell: '617-555-1761'
    }
  }
}

```

```
    },  
    sex: 'Male'  
  }  
};  
db.update(json, 'object');
```

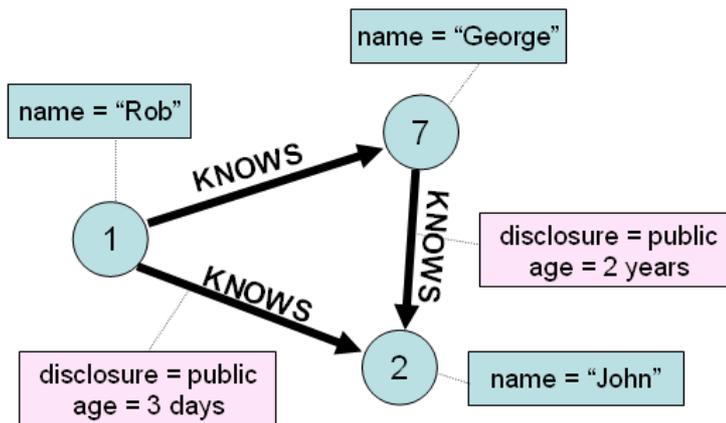
and retrieve it as follows:

```
var json = db.retrieve({global: 'person'}, "object").object;
```

4) Graph Databases

NoSQL graph databases such as Neo4j are used to represent complex networks of relationships in terms of nodes and relationships between nodes (*aka* "edges"), with key/value pairs attached to both nodes and relationships.

The classic use for graph databases is to represent social networks. Take the following example:



This can be represented using Globals as:

```
person[personId, "knows", personId] == ""
person[personId, "knows", personId, key) == value
person[personId, "name"] == name
```

The age of the "knows" relationship would probably be derived from a timestamp key that is created when the relationship is first saved.

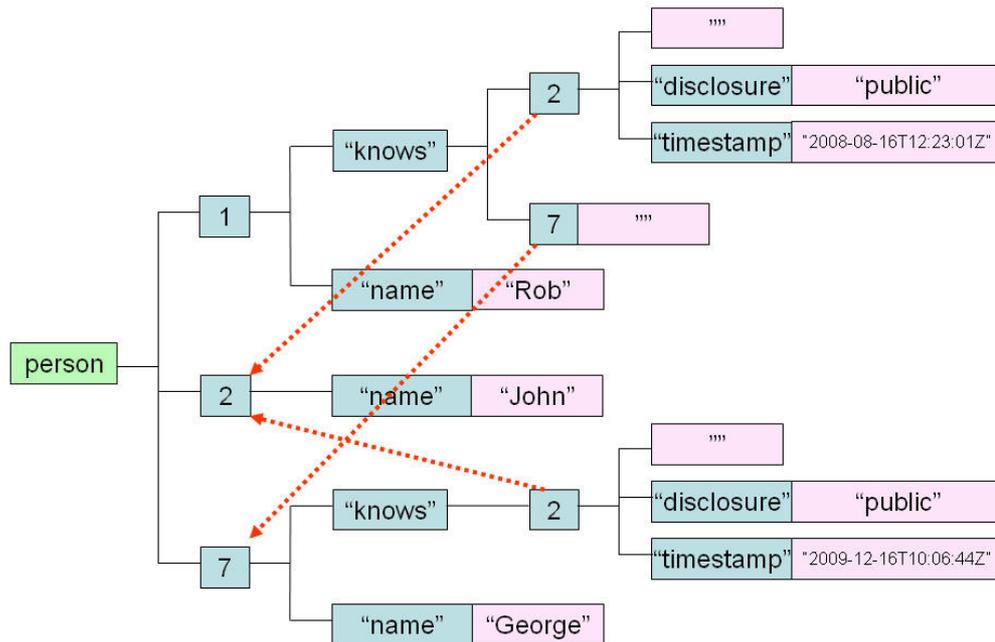
eg:

```
person[1, "knows", 2] == ""
person[1, "knows", 2, "disclosure"] == "public"
person[1, "knows", 2, "timestamp"] == "2008-08-16T12:23:01Z"
person[1, "knows", 7] == ""
person[1, "name"] == "Rob"

person[2, "name"] == "John"

person[7, "knows", 2] == ""
person[7, "knows", 2, "disclosure"] == "public"
person[7, "knows", 2, "timestamp"] == "2009-12-16T10:06:44Z"
person[7, "name"] == "George"
```

or diagrammatically (the red dotted lines show the implied relationships in this model):



In fact, for a generic graph database, the model would be further abstracted to represent nodes and vertices, for example something like:

```
node[nodeType, nodeId] == ""
node[nodeType, nodeId, attribute] == attributeValue

edge[edgeType, fromNodeId, toNodeId] == ""
edge[edgeType, fromNodeId, toNodeId, attribute] == attributeValue

edgeReverse[edgeType, toNodeId, fromNodeId] == ""
```

So you can see that the flexibility of Globals and their sparse nature lends themselves very naturally and simply to defining complex graph databases.

5) Other Database Models

Globals aren't just restricted to the NoSQL models. They can also be used to model further database types:

- **XML DOM/ Native XML Database.** GlobalsDB is a perfect vehicle for implementing a persistent XML DOM. It is essentially a Graph structure that represents the nodes (and their associated type) and the relationships between them (eg firstChild, lastChild, nextSibling, parent etc). In essence, such an implementation would allow GlobalsDB to behave as a Native XML Database. The ewdDOM Node.js module (<https://github.com/robtweed/ewdDOM>) is one such lightweight implementation.
- **Relational tables.** Caché models relational tables onto Globals. It is then possible to use natively-implemented SQL-based queries. So, whilst GlobalsDB can be thought of as the basis of a NoSQL engine, Caché adds the capabilities of a NOSQL (ie Not-Only SQL) database.
- **Persistent Object Storage.** Caché goes still further and models Objects onto the underlying Global storage, and provides a direct mapping between those objects and relational tables. Hopefully by now you can probably envisage how this might be achieved.

The really interesting thing about GlobalsDB is that, unlike the commonly-known NoSQL databases, it isn't shoe-horned into one particular category, but it can have multiple, simultaneous characteristics. So a GlobalsDB system could support any or all of the database types described above, simultaneously if required. So it's like having Redis, CouchDB, SimpleDB, Neo4j and a Native XML Database all running in the same database, all at the same time!

If you're interested in Node.js and NoSQL databases, you really need to take a look at GlobalsDB. It truly is a Universal NoSQL Engine!

Conclusion

There's more to using Globals than just the description we've given. However, hopefully this summary overview has demonstrated that they are a nice means of abstraction that is flexible and makes modelling many different use cases very easy. The secret sauce, of course, is their implementation. If done correctly, there are smart design choices that really make for astounding performance.