Article
Sergey Kamenev · Jun 14, 2017 10m read

# Globals - Magic swords for storing data. Trees. Part 2

*Beginning - see Part 1.*

## 3. Variants of structures when using globals

A structure, such as an ordered tree, has various special cases. Let's take a look at those that have practical value for working with globals.

### 3.1 Special case 1. One node without branches

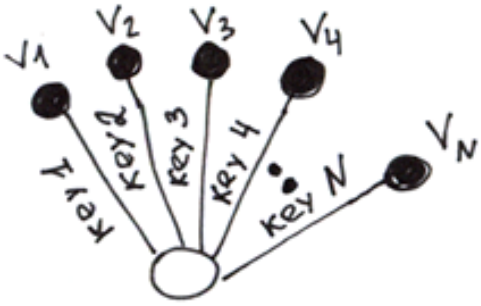

Globals can be used not just like an array, but like regular variables. For instance, for creating a counter:

```
Set ^counter = 0  ; setting counter
Set id=$Increment(^counter) ;  atomic incrementation
```

At the same time, a global can have branches in addition to its value. One doesn't exclude the other.

## 3.2 Special case 2. One node and multiple branches

In fact, it's a classic key-value base. And if we save tuples of values instead of values, we'll get a regular table with a primary key.




*Key-Value DB on the Global*

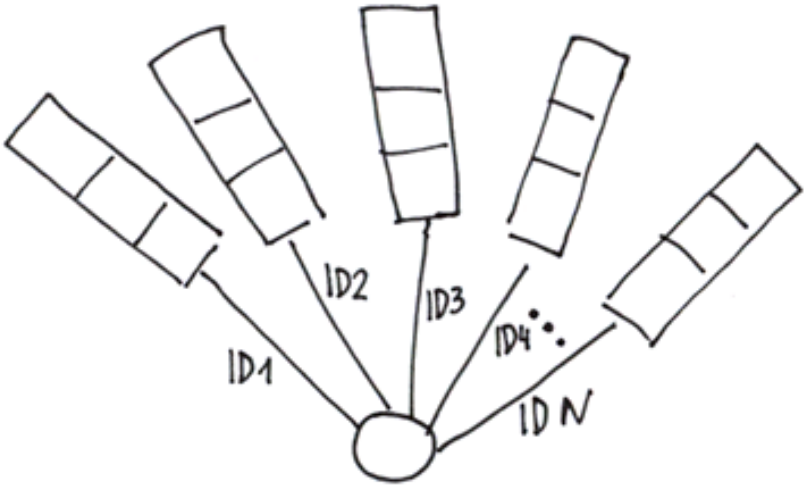



*Table on the Global.*
*Storing row as Value.*

In order to implement a table based on globals, we will have to form strings from column values, then save them to a global by the primary key. In order to be able to split the string into columns during reading, we can use the following:

1. Delimiter characters.

```
Set ^t(id1) = "col11/col21/col31"
Set ^t(id2) = "col12/col22/col32"
```

2. A fixed scheme, whereby each field occupies a particular number of bytes. This is how it's usually done in relational databases.
3. A special $LB function (introduced in Caché) that composes a string from values.

```
Set ^t(id1) = $LB("col11", "col21", "col31")
Set ^t(id2) = $LB("col12", "col22", "col32")
```

What's interesting is that it's not hard to do something similar to foreign keys in relational DB's using globals. Let's call such structures index globals. An index global is a supplementary tree for quick searching by fields that are not an integral part of the primary key of the main global. You need to write additional code to fill and use it.

Let's create an index global based on the first column.

```
Set ^i("col11", id1) = 1
Set ^i("col12", id2) = 1
```

To search quickly by the first column, you will need to look into the ^i global and find primary keys (id) corresponding to the necessary value in the first column.

When inserting a value, we can create both values and index globals for the necessary fields. For reliability, let's wrap it into a transaction.

```
TSTART
Set ^t(id1) = $LB("col11", "col21", "col31")
Set ^i("col11", id1) = 1
TCOMMIT
```

More information on making tables in M using globals and emulation of secondary keys.

These tables will work just as fast as in traditional DB's (or even faster) if insert/update/delete functions are written in COS/M and compiled.

I verified this statement by applying a bulk of INSERT and SELECT operations to a single two-column table, also using TSTART and TCOMMIT command (transactions).

I have not tested more complex scenarios with concurrent access and parallel transactions.

Without using transactions, the speed of insertion for a million values was 778,361 inserts/sec.
For 300 million values, the speed was 422,141 inserts/second.

When transactions were used, the speed reached 572,082 inserts/second for 50 million values. All operations were run from compiled M-code. I used regular hard drives, not SSD's. RAID5 with Write-back. All running on a Phenom II 1100T CPU.

To conduct the same test for an SQL database, we would need to write a stored procedure that will make inserts in a loop. When testing MySQL 5.5 (InnoDB storage) using the same method, I never got more than 11K inserts per second.

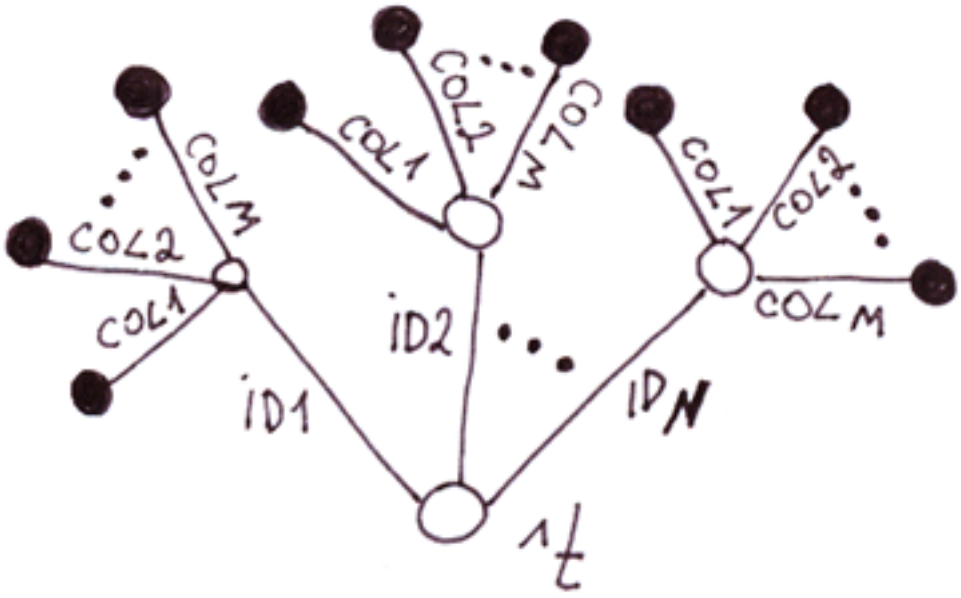
Right, implementation of tables with globals is more complex than doing the same in relational databases. That is why industrial DB's based on globals have SQL access for simplified work with tabular data.


In general, if the data schema is not going to change often, the speed of insertion in not critical and the entire database can be easily represented with normalized tables, it's easier to work with SQL, since it provides a higher level of abstraction.


In this case, I wanted to show that globals can be used as a constructor for creating other DB's. Like the assembler language that can be used for creating other languages. And here are some examples of using globals for creating counterparts of key-values, lists, sets, tabular, document-oriented DB's.

If you need to create a non-standard DB with minimal efforts, you should consider using globals.

## 3.3 Special case 3. A two-level tree with each second-level node having a fixed number of branches




You have probably guessed: it's an alternative implementation of tables using globals. Let's compare it with the previous one.

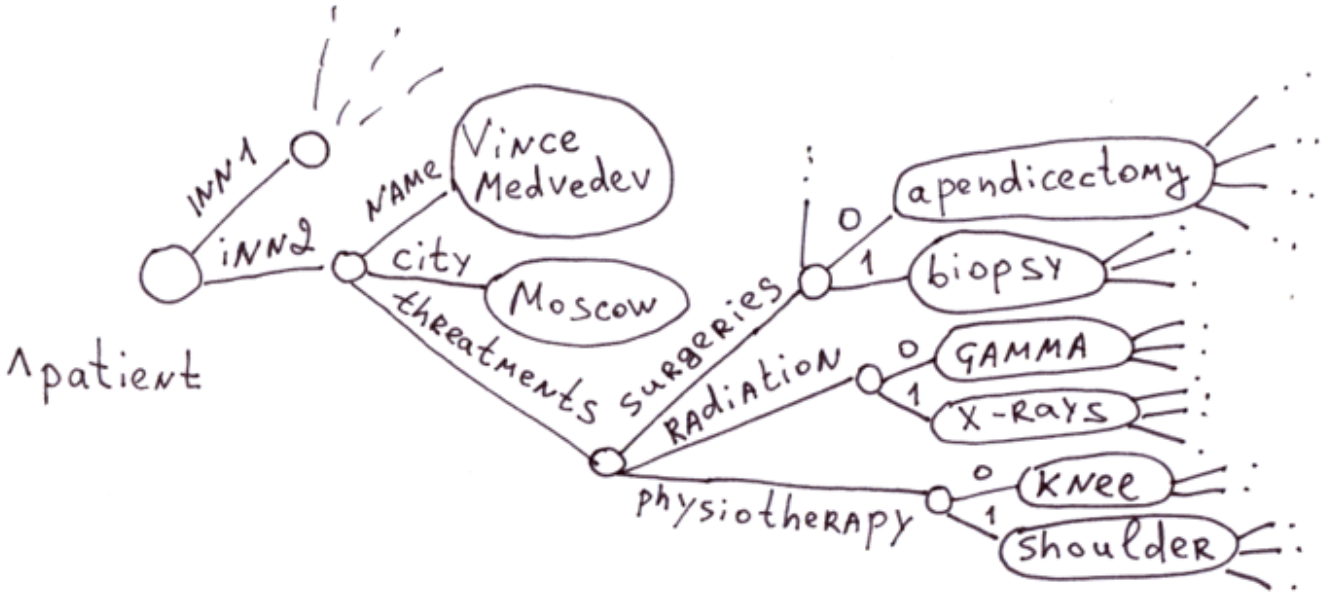
| Tables in a two-level tree vs. Tables in a one-level tree. | |
|---|---|
| Cons | Pros |
| 1. Slower inserts, since the number of nodes must be set equal to the number of columns.<br>2. Higher hard drive space consumption, since global indexes (like array indexes) with column names occupy space on the hard drive and are duplicated for each row. | 1. Faster access to values of particular columns, since you don't need to parse the string. According to my tests, it's 11.5% faster for 2 columns and even faster for more columns.<br>2. Easier to change the data schema<br>3. Easier to read code |

**Conclusion:** nothing to write home about. Since performance is one of the key advantages of globals, there is virtually no point in using this approach, since it's unlikely to work faster than regular tables in relational databases.

## 3.4 General case. Trees and ordered keys

Any data structure that can be represented as a tree fits globals in a perfect way.

### 3.4.1 Objects with sub-objects




This is the area where globals are traditionally used. There are numerous illnesses, drugs, symptoms and treatment methods in the medical area. It's irrational to create a table with a million fields for every patient, especially since 99% of them will be blank.

Imagine an SQL DB comprised of the following tables: "Patient" ~100,000 fields, "Medication" 100,000 fields, "Therapy" 100,000 fields, "Complications" 100,000 fields and so on. As an alternative, you can create a DB with thousands of tables, each for a particular patient type (and they can overlap, too!), treatment, medication, as well as thousands of tables for relations between these tables.

Globals fit healthcare like a glove, since they make it possible for each patient to have a complete case record, list of therapies, administered drugs and their effects - all in the form of a tree, without wasting too much disk space on empty columns, as would be the case with relational databases.


**Globals work well for databases with personal details,** when the task is to accumulate and systemize the maximum of various personal data about a client. This is especially important for healthcare, banking, marketing, archiving and other areas.

It goes without saying that SQL also enables you to emulate a tree using just several tables (EAV, 1,2,3,4,5,6, 7,8), but it's a lot more complex and works slower. In essence, we'd have to write a global based on tables and hide all table-related routines under an abstraction layer. It's not correct to emulate a lower-level technology (globals) with the help of a higher-level one (SQL). It's just unjustified.

It's not a secret that changing a data schema in gigantic tables (ALTER TABLE) may take a considerable amount of time. MySQL, for example, performs the ALTER TABLE ADD|DROP COLUMN operation by copying all the data from the old table to the new one (I tested it on MyISAM and InnoDB). Which may hang up a production database with billions of records for days, if not weeks.


**If we are using globals, changing the data structure comes at no cost to us.** We can add any new

properties to any object on any level of the hierarchy at any given time. Changes that require branches to be renamed can be applied in the background mode with the DB up and running.

Therefore, when it comes to storing objects with a large number of optional properties, globals work perfectly well.

Let me remind you that access to any of the properties is instant, since in a global, all paths are a B-tree.

In the general case, databases based on globals are a type of document-oriented databases that support the storing of hierarchical information. Therefore, document-oriented databases can efficiently compete with globals in the field of storing medical cards.

But's it's not quite it yet

Let's take MongoDB, for example. **In this field**, it loses to globals for the following reasons:

1. **Document size.** The storage unit is a text in the JSON format (BSON, to be exact) with a maximum size of around 16 MB. The limitation was introduced on purpose to make sure that the JSON database doesn't get too slow during parsing, when a huge JSON document is saved to it and particular field values are addressed. This document is supposed to have complete information about a patient. We all know how thick patient cards can be. If the maximum size of the card is capped at 16 MB, it immediately filters out patients whose cards contain MRI scans, X-ray scans and and other materials. A single branch of a global can have gigabytes and petabytes of terabytes of data. It sort of says it all, but let me tell you more.
2. **The time required for creating/changing/removing new properties from the patient card.** Such a database would need to copy the entire card into the memory (lots of data!), parse the BSON data, add/change/remove the new node, update indexes, pack it all back to BSON and save to the disk. A global would only need to address the necessary property and perform the necessary operation.
3. **Speed of access to particular properties.** If the document has many properties and a multi-level structure, access to particular properties will be faster because each path in the global is a B-tree. In BSON, you will need to linearly parse the document to find the necessary property.

### 3.3.2 Associative arrays

Associative arrays (even with nested arrays) work perfectly with globals. For example, this PHP array will look like the first illustration in 3.3.1.

```
$a = array(
  "name" => "Vince Medvedev",
  "city" => "Moscow",
  "threatments" => array(
    "surgeries" => array("apedicectomy", "biopsy"),
```

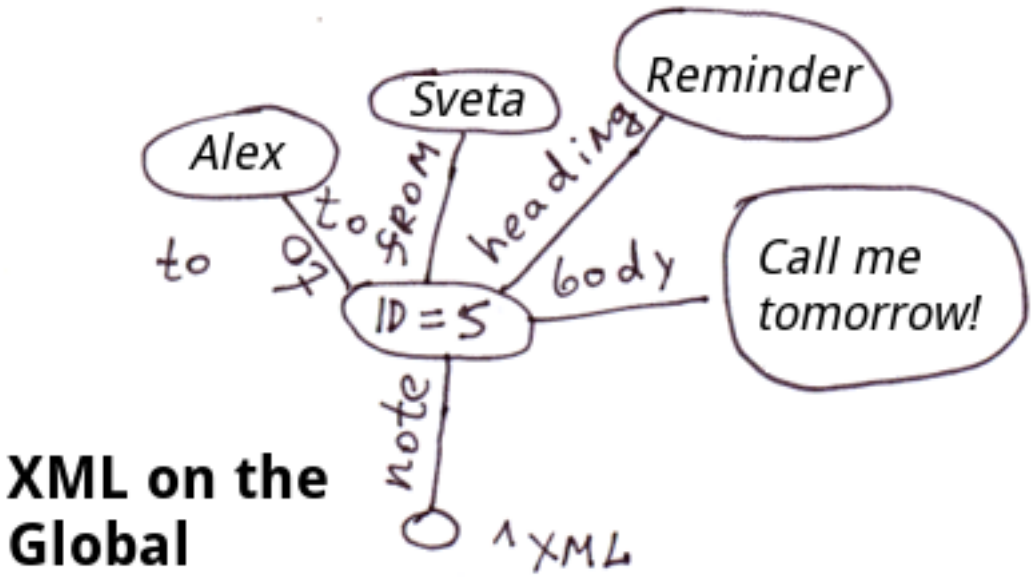
```
    "radiation" => array("gamma", "x-rays"),
    "physiotherapy" => array("knee", "shoulder")
  )
);
```

#### 3.3.3 Hierarchical documents: XML, JSON

Can be also easily stored in globals and decomposed in different ways.

**XML**

The easiest method of decomposing XML into globals is to store tag attributes in nodes. And if you need quick access to tag attributes, we can place them in separate branches.

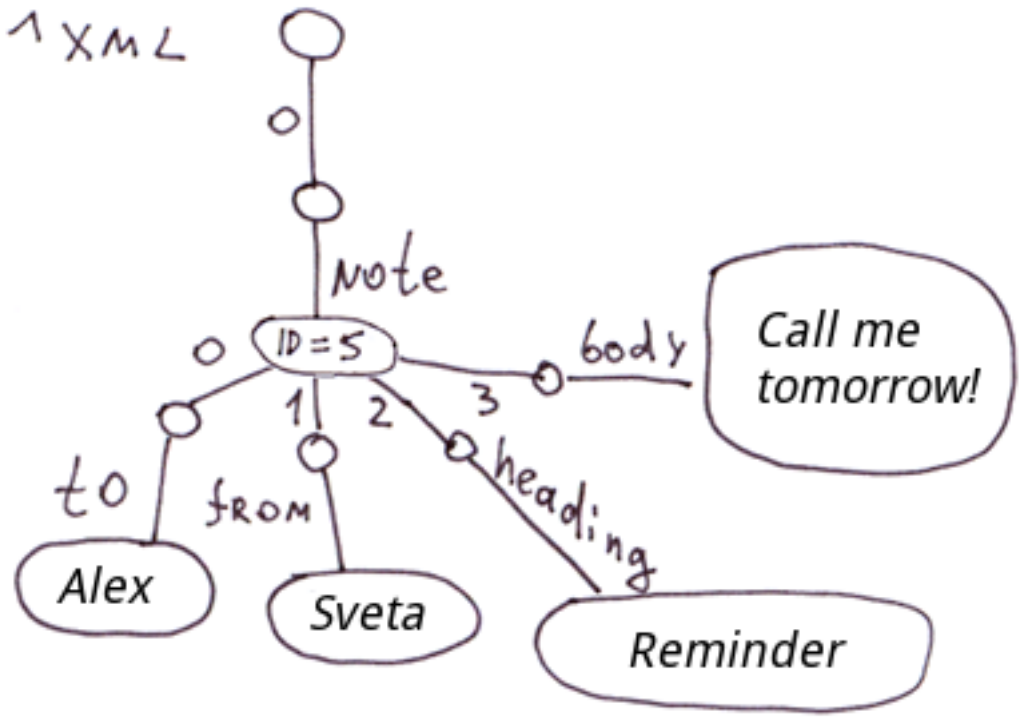



```
<note id=5>
<to>Alex</to>
<from>Sveta</from>
<heading>Reminder</heading>
<body>Call me tomorrow!</body>
</note>
```

In COS, the code will look like this:

```
Set ^xml("note")="id=5"
Set ^xml("note","to")="Alex"
Set ^xml("note","from")="Sveta"
Set ^xml("note","heading")="Reminder"
Set ^xml("note","body")="Call me tomorrow!"
```

**Note:** For XML, JSON and associative arrays, you can come up with a number of methods of displaying them in globals. In this particular case, we did not reflect the order of nested tags in the "note" tag. In the **^xml** global, nested tags will be displayed in the alphabetical order. For precise display of order, you can use the following model, for example:




**JSON.**

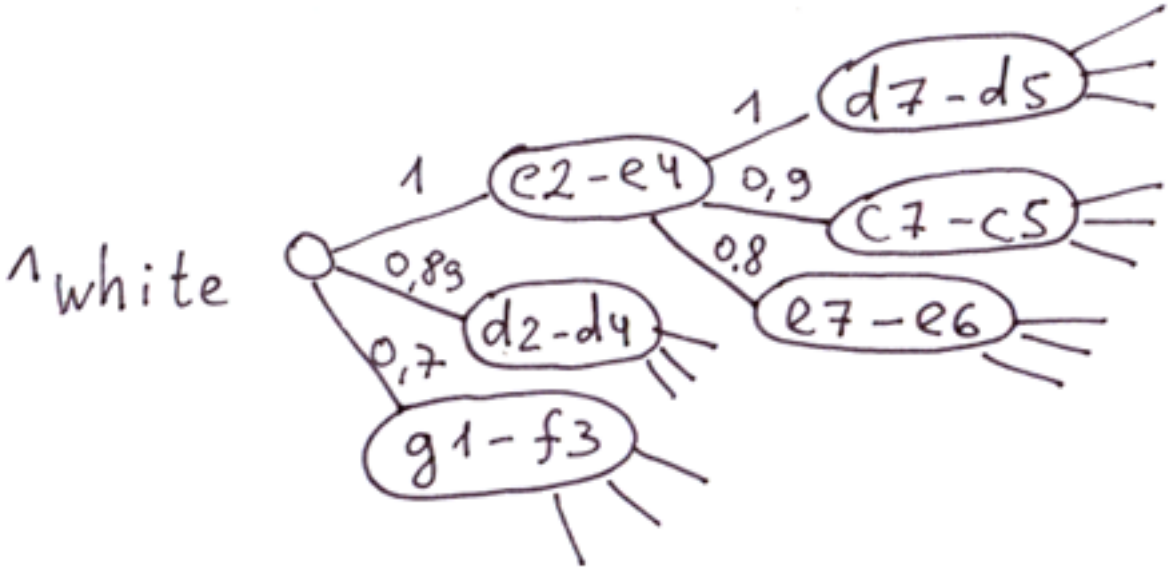
The content of this JSON document is shown in the first illustration in Section 3.3.1:

```
var document = {
  "name": "Vince Medvedev",
  "city": "Moscow",
  "threatments": {
    "surgeries": ["apedicectomy", "biopsy"],
    "radiation": ["gamma", "x-rays"],
    "physiotherapy": ["knee", "shoulder"]
  },
};
```

### 3.3.4 Identical structures bound by hierarchical relations

Examples: structure of sales offices, positions of people in an MLM structure, base of chess debuts.

**Database of debuts.** You can use a move strength assessment as the value of a global's node index. In this case, you will need to select a branch with the highest weight to determine the best move. In the global, all branches on every level will be sorted by the move strength.




*An example of debuts DB on the Global. Select the branch with the highest weight and make a move.*

**The structure of sales offices, people in an MLM company.** Nodes can store some caching values reflecting the characteristics of the entire sub-tree. For example, the sales of this particular sub-tree. We can get exact information about the achievements of any branch at any moment.


## 4. Situations where using globals pays off

The first column contains a list of cases where using globals will give you a considerable advantage in terms of performance, and the second one - a list of situations where they will simplify development or the data model.

| Speed | Convenience of data processing/presentation |
|---|---|
| 1. Insertion [with automatic sorting on each level], [indexing by the primary key]<br>2. Subtree removal<br>3. Objects with lots of nested properties that you need individual access to<br>4. A hierarchical structure with a possibility of traversing child branches starting from any branch, even a non-existing one<br>5. In-depth tree traversal | 1. Objects/instances with a huge number of non-required [an/or nested] properties/instances<br>2. Schema-less data - new properties can often be added and old ones removed.<br>3. You need to create a non-standard DB.<br>4. Path databases and solution trees. When paths can be conveniently represented as a tree..<br>5. Removal of hierarchical structures without using recursion |

*To be continued in* "Globals - Magic swords for Storing Data. Sparse Arrays. Part 3"

**Disclaimer:** *this article and my comments on it reflect my opinion only and have nothing to do with the official position of the InterSystems Corporation.*

#Beginner #Data Model #Globals #Node.js #Performance #Relational Tables #Caché #InterSystems IRIS