Vyacheslav FOLLOW
Experienced Software Engineer, with Project Management Experience

Storing Tree Structures in MongoDB: Code Examples

Published Jun 12, 2015

This is an educational article demonstrating approaches for storing tree structures with NoSQL database, MongoDB.

Background

In a real life, almost any project deals with the tree structures.

Different kinds of taxonomies, site structures, etc,
require modeling of hierarchy relations. In this MongoDB tutorial, I will illustrate how to use five typical approaches (plus one combination of operating with hierarchy data) on an example dataset.

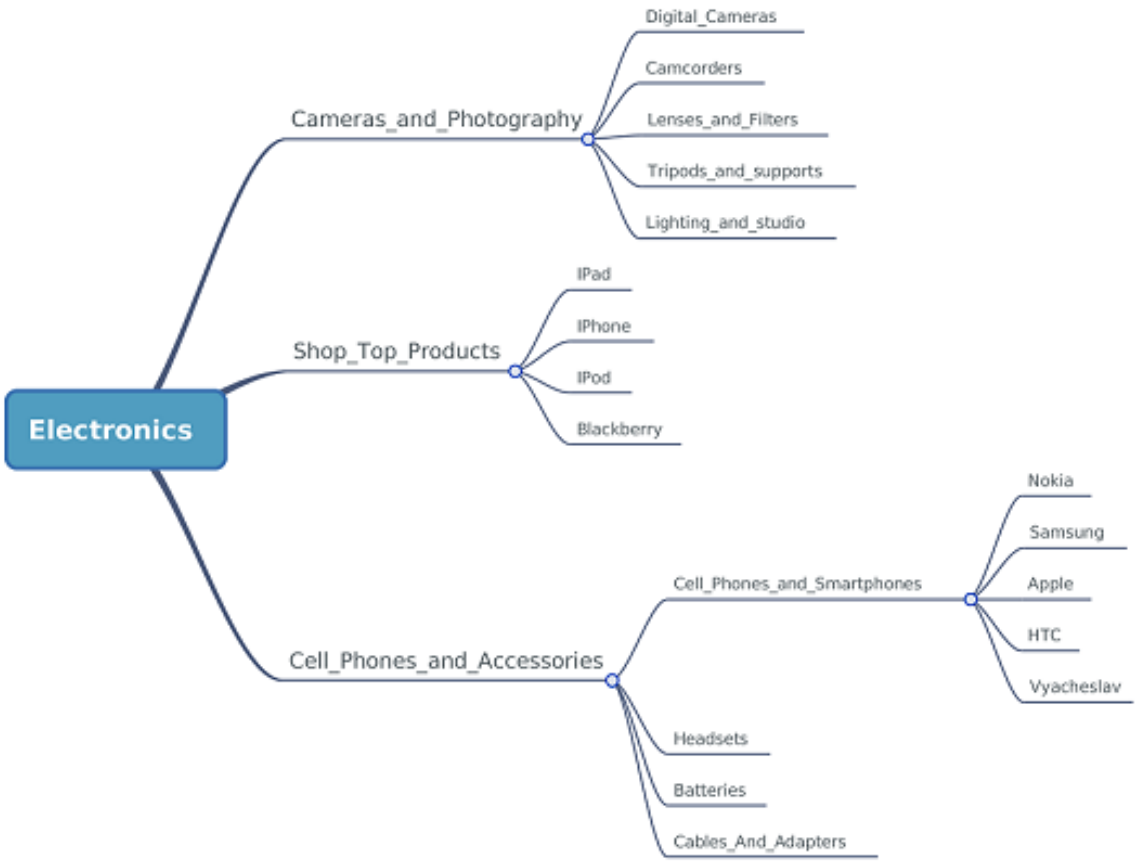
Those approaches are:

- Model Tree Structures with Child References

- Model Tree Structures with an Array of Ancestors
- Model Tree Structures with Materialized Paths
- Model Tree Structures with Nested Sets

Note: *This article is inspired by another article 'Model Tree Structures in MongoDB' by MongoDB, but does not copy it. Instead, this tutorial provides additional examples on typical operations with tree management. Please refer to the MongoDB article to get a more solid understanding of the approach.*

At any rate, I'll be using some fake e-shop's goods taxonomy for the demo dataset.



In a typical site scenario, we should be able to:

- Operate within the tree (insert new node under specific parent, update/remove existing node, move node across the tree, etc.)
- Get the path to a node (in order to build the breadcrumb section, for example)
- Get all node descendants (e.g. to be able to select goods from a more general category like 'Cell Phones and Accessories', which should include goods from all subcategories.)

The respective example situations we will work on are how to:

- Add a new node called `LG` under electronics
- Move the `LG` node under a `Cell_Phones_And_Smartphones` node
- Remove the `LG` node from the tree
- Get children nodes of an `Electronics` node
- Get the path to a `Nokia` node
- Get all descendants of the `Cell_Phones_and_Accessories` node

Please refer to the image above for a more visual representation of the data set.

Tree Structure with Parent Reference

This is the most commonly used approach. For each node, we'd store the **ID**, **ParentReference**, and **Order**:

ID	Parent	Order
Cell_Phones_and_Smartphones	Cell_Phones_and_Accessories	10
LTG	Cell_Phones_and_Smartphones	10

Operating with the Tree

This is pretty simple, but changing the position of the node within siblings will require additional calculations.
You might want to set high numbers like `item position * 10^6` for the order so you can set a new node order as `trunc (lower sibling order - higher sibling order)/2` - this will give you enough operations until you need to traverse whole the tree and set the order defaults to big numbers again.

Adding a New Node

```
var existingelemscount = db.categoriesPCO.find({parent:'Electronics'}).count();
var neworder = (existingelemscount+1)*10;
db.categoriesPCO.insert({_id:'LG', parent:'Electronics', someadditionalattr:'te
//{ "_id" : "LG", "parent" : "Electronics", "someadditionalattr" : "test", "ord
```

Updating/Moving a Node

```
existingelemscount = db.categoriesPCO.find({parent:'Cell_Phones_and_Smartphones
neworder = (existingelemscount+1)*10;
db.categoriesPCO.update({_id:'LG'},{$set:{parent:'Cell_Phones_and_Smartphones',
//{ "_id" : "LG", "order" : 60, "parent" : "Cell_Phones_and_Smartphones", "some
```

Removing a Node

```
db.categoriesPCO.remove({_id:'LG'});
```

```
db.categoriesPCO.find({$query:{parent:'Electronics'}, $orderby:{order:1}})
//{ "_id" : "Cameras_and_Photography", "parent" : "Electronics", "order" : 10 }
//{ "_id" : "Shop_Top_Products", "parent" : "Electronics", "order" : 20 }
//{ "_id" : "Cell_Phones_and_Accessories", "parent" : "Electronics", "order" :
```

Getting all Node Descendants

Unfortunately, also involves recursive operation

```
var descendants=[]
var stack=[];
var item = db.categoriesPCO.findOne({_id:"Cell_Phones_and_Accessories"});
stack.push(item);
while (stack.length>0){
    var currentnode = stack.pop();
    var children = db.categoriesPCO.find({parent:currentnode._id});
    while(true === children.hasNext()) {
        var child = children.next();
        descendants.push(child._id);
        stack.push(child);
    }
}


descendants.join(",")
//Cell_Phones_and_Smartphones,Headsets,Batteries,Cables_And_Adapters,Nokia,Sams
```

Getting the Path to a Node

Unfortunately, this involves recursive operations:

```
var path=[]
var item = db.categoriesPCO.findOne({_id:"Nokia"})
while (item.parent !== null) {
    item=db.categoriesPCO.findOne({_id:item.parent});
    path.push(item._id);
}

path.reverse().join(' / ');
//Electronics / Cell_Phones_and_Accessories / Cell_Phones_and_Smartphones
```

Indexes

Recommended index is on fields parent and order

```
db.categoriesPCO.ensureIndex( { parent: 1, order:1 } )
```

Tree Structure with Child Reference

For each node, we'll store the **ID** and **ChildReferences**.

ID	Childs
Cell_Phones_and_Smartphones	"Nokia","Samsung","Apple","HTC","Ukrtelecom"
HTC	

Please note that in this case we do not need an order field, because the child's collection
already provides this information. Most languages respect the array order. If this is not in case for your programming language, you might consider writing additional code to preserve the order, but this will also make things more complicated

Adding a New Node

```
db.categoriesCRO.insert({_id:'LG', childs:[]});
db.categoriesCRO.update({_id:'Electronics'},{  $addToSet:{childs:'LG'}});
//{ "_id" : "Electronics", "childs" : [         "Cameras_and_Photography",
```

Updating/Moving a Node

Rearranging order under the same parent:

```
db.categoriesCRO.update({_id:'Electronics'},{$set:{"childs.1":'LG',"childs.3":'
//{ "_id" : "Electronics", "childs" : [         "Cameras_and_Photography",
```

Moving a Node:

```
db.categoriesCRO.update({_id:'Cell_Phones_and_Smartphones'},{  $addToSet:{child
db.categoriesCRO.update({_id:'Electronics'},{$pull:{childs:'LG'}});
//{ "_id" : "Cell_Phones_and_Smartphones", "childs" : [ "Nokia", "Samsung", "Ap
```

Removing a Node

```
db.categoriesCRO.update({_id:'Cell_Phones_and_Smartphones'},{$pull:{childs:'LG'
db.categoriesCRO.remove({_id:'LG'});
```

Note: this requires additional client-side sorting in the parent array sequence

```
var parent = db.categoriesCRO.findOne({_id:'Electronics'})
db.categoriesCRO.find({_id:{$in:parent.childs}})
```

Result:

```
{ "_id" : "Cameras_and_Photography", "childs" : [      "Digital_Cameras",
{ "_id" : "Cell_Phones_and_Accessories", "childs" : [  "Cell_Phones_and_Smartph
{ "_id" : "Shop_Top_Products", "childs" : [ "IPad", "IPhone", "IPod", "Blackber

//parent:
{
  "_id" : "Electronics",
  "childs" : [
    "Cameras_and_Photography",
    "Cell_Phones_and_Accessories",
    "Shop_Top_Products"
  ]
}
```

As you can see, we have ordered array childs, which can be used to sort the result set on a client.

Getting all Node Descendants

```
var descendants=[]
var stack=[];
var item = db.categoriesCRO.findOne({_id:"Cell_Phones_and_Accessories"});
stack.push(item);
while (stack.length>0){
    var currentnode = stack.pop();
    var children = db.categoriesCRO.find({_id:{$in:currentnode.childs}});

    while(true === children.hasNext()) {
        var child = children.next();
        descendants.push(child._id);
        if(child.childs.length>0){
          stack.push(child);
        }
    }
}

//Batteries,Cables_And_Adapters,Cell_Phones_and_Smartphones,Headsets,Apple,HTC,
descendants.join(",")
```

Getting the Path to a Node

```
var path=[]
var item = db.categoriesCRO.findOne({_id:"Nokia"})
while ((item=db.categoriesCRO.findOne({childs:item._id}))) {
    path.push(item._id);
}

path.reverse().join(' / ');
//Electronics / Cell_Phones_and_Accessories / Cell_Phones_and_Smartphones
```sql

### Indexes
Recommended index is putting index on childs:
```sql
db.categoriesCRO.ensureIndex( { childs: 1 } )
```

Tree Structure with an Array of Ancestors

For each node we'll store the **ID**, **ParentReference**, and **AncestorReferences** as in the example below:

ID	Parent	Ancestors
Cell_Phones_and_Smartphones	Cell_Phones_and_Accessories	"Electronics","Cell_Phones_and_Accessories"
HTC	Cell_Phones_and_Smartphones	"Electronics","Cell_Phones_and_Accessories","Cell_Phones_and_Smartphones"

Adding a New Node

```
var ancestorpath = db.categoriesAAO.findOne({_id:'Electronics'}).ancestors;
ancestorpath.push('Electronics')
db.categoriesAAO.insert({_id:'LG', parent:'Electronics',ancestors:ancestorpath}
//{ "_id" : "LG", "parent" : "Electronics", "ancestors" : [ "Electronics" ] }
```

Updating/Moving a Node

Moving the node:

```
ancestorpath = db.categoriesAAO.findOne({_id:'Cell_Phones_and_Smartphones'}).an
ancestorpath.push('Cell_Phones_and_Smartphones')
db.categoriesAAO.update({_id:'LG'},{$set:{parent:'Cell_Phones_and_Smartphones',
//{ "_id" : "LG", "ancestors" : [       "Electronics",  "Cell_Phones_and_Accesso
```

Removing a Node

```
db.categoriesAAO.remove({_id:'LG'});
```

Note: unless you introduce an order field, it is impossible to get an ordered list of node children. You should consider another approach if you need it to be ordered.

```
db.categoriesAAO.find({$query:{parent:'Electronics'}})
```

Getting all Node Descendants

There are two options to get all node descendants. One is a classic approach through recursion:

```
var ancestors = db.categoriesAAO.find({ancestors:"Cell_Phones_and_Accessories"}
while(true === ancestors.hasNext()) {
       var elem = ancestors.next();
       descendants.push(elem._id);
   }
descendants.join(",")
//Cell_Phones_and_Smartphones,Headsets,Batteries,Cables_And_Adapters,Nokia,Sams
```

The other is to use an aggregation framework introduced in MongoDB 2.2:

```
var aggrancestors = db.categoriesAAO.aggregate([
    {$match:{ancestors:"Cell_Phones_and_Accessories"}},
    {$project:{_id:1}},
    {$group:{_id:{},ancestors:{$addToSet:"$_id"}}}
])

descendants = aggrancestors.result[0].ancestors
descendants.join(",")
//Vyacheslav,HTC,Samsung,Cables_And_Adapters,Batteries,Headsets,Apple,Nokia,Cel
```

Tree Structure with Materialized Path

For each node, we'll store the **ID** and **PathToNode**

ID	Ancestors
Cell_Phones_and_Smartphones	"Electronics,Cell_Phones_and_Accessories"
HTC	"Electronics,Cell_Phones_and_Accessories,Cell_Phones_and_Smartphones"

Adding a New Node

```
var ancestorpath = db.categoriesMP.findOne({_id:'Electronics'}).path;
ancestorpath += 'Electronics,'
db.categoriesMP.insert({_id:'LG', path:ancestorpath});
//{ "_id" : "LG", "path" : "Electronics," }
```

Updating/Moving a Node

Moving the Node

```
ancestorpath = db.categoriesMP.findOne({_id:'Cell_Phones_and_Smartphones'}).pat
ancestorpath +='Cell_Phones_and_Smartphones,'
db.categoriesMP.update({_id:'LG'},{$set:{path:ancestorpath}});
//{ "_id" : "LG", "path" : "Electronics,Cell_Phones_and_Accessories,Cell_Phones
```

Removing a Node

```
db.categoriesMP.remove({_id:'LG'});
```

Getting the Node Children (unordered):

Note: unless you introduce the order field, it is impossible to get ordered list of node children. You should consider another approach if you need order.

```
db.categoriesMP.find({$query:{path:'Electronics,'}})
//{ "_id" : "Cameras_and_Photography", "path" : "Electronics," }
//{ "_id" : "Shop_Top_Products", "path" : "Electronics," }
//{ "_id" : "Cell_Phones_and_Accessories", "path" : "Electronics," }
```

Getting All Node Descendants

Single select, regexp starts with `^`, which allows using the index for matching

```
var descendants=[]
var item = db.categoriesMP.findOne({_id:"Cell_Phones_and_Accessories"});
var criteria = '^'+item.path+item._id+',';
var children = db.categoriesMP.find({path: { $regex: criteria, $options: 'i' }}
while(true === children.hasNext()) {
  var child = children.next();
  descendants.push(child._id);
}


descendants.join(",")
//Cell_Phones_and_Smartphones,Headsets,Batteries,Cables_And_Adapters,Nokia,Sams
```

Getting the Path to Node

We can obtain the path directly from node without issuing additional selects.

```
var path=[]
var item = db.categoriesMP.findOne({_id:"Nokia"})
print (item.path)
//Electronics,Cell_Phones_and_Accessories,Cell_Phones_and_Smartphones,
```

Indexes

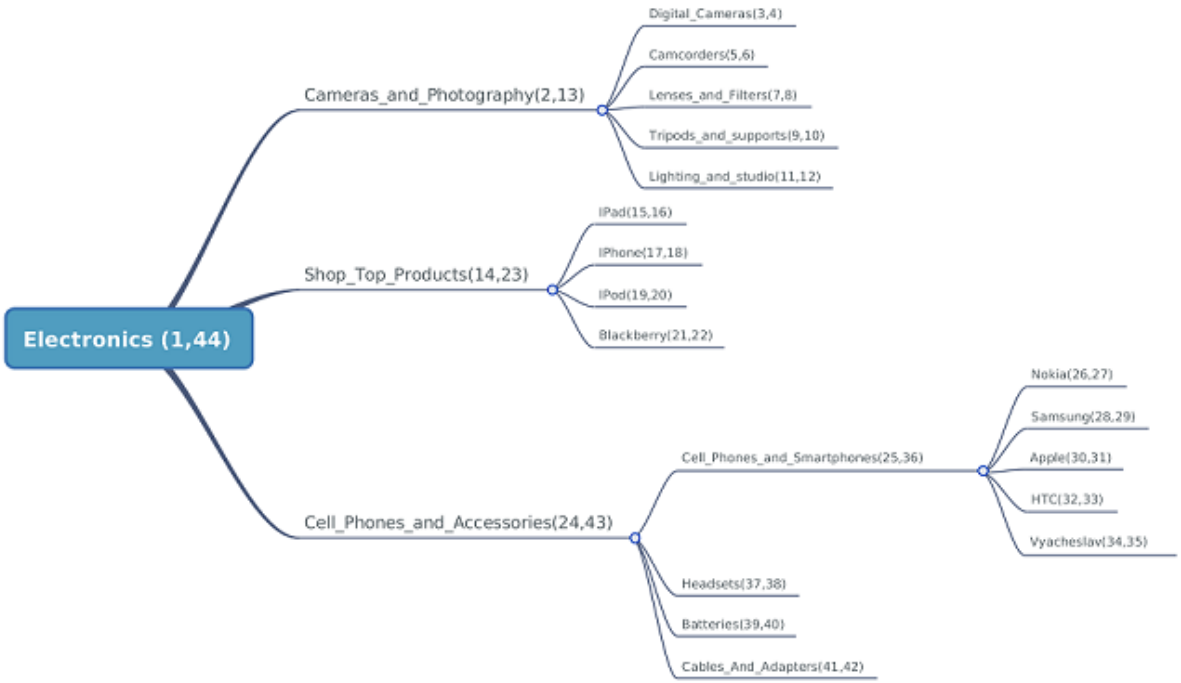
The recommended index is putting index on path

```
  db.categoriesAAO.ensureIndex( { path: 1 } )
```

Tree Structure with Nested Sets

ID	Left	Right
Cell_Phones_and_Smartphones	25	36
HTC	32	33

The left field also can be treated as an order field



Adding a New Node

Please refer to image above. Assume that we want to insert the `LG` node after `shop_top_products(14,23)` .

A new node would have a left value of `24` , which will affect all the remaining left values according to traversal rules. It will also have a right value of `25` ,

Steps to Add a Node:

- take the next node in a traversal tree
- the new node will have a left value of the following sibling, and the right value will be incremented by the two following siblings' left one
- now we have to create the place for the new node. Update affects right values of all ancestor nodes and also affects all nodes that remain for traversal
- Only after creating place new node can be inserted

```
var followingsibling = db.categoriesNSO.findOne({_id:"Cell_Phones_and_Accessori

var newnode = {_id:'LG', left:followingsibling.left,right:followingsibling.left

db.categoriesNSO.update({right:{$gt:followingsibling.right}},{$inc:{right:2}},

db.categoriesNSO.update({left:{$gte:followingsibling.left}, right:{$lte:followi

db.categoriesNSO.insert(newnode)
```

Let's check the result:

```
+-Electronics (1,46)
    +---Cameras_and_Photography (2,13)
          +------Digital_Cameras (3,4)
          +------Camcorders (5,6)
          +------Lenses_and_Filters (7,8)
          +------Tripods_and_supports (9,10)
          +------Lighting_and_studio (11,12)
      +----Shop_Top_Products (14,23)
          +------IPad (15,16)
          +------IPhone (17,18)
          +------IPod (19,20)
          +------Blackberry (21,22)
      +----LG (24,25)
      +----Cell_Phones_and_Accessories (26,45)
          +------Cell_Phones_and_Smartphones (27,38)
                +---------Nokia (28,29)
                +---------Samsung (30,31)
                +---------Apple (32,33)
                +---------HTC (34,35)
                +---------Vyacheslav (36,37)
            +-------Headsets (39,40)
            +-------Batteries (41,42)
            +-------Cables_And_Adapters (43,44)
```

Removing a Node

While rearranging node order within same parent is potentially identical to exchanging node's left and right values,
the formal way of moving the node is to first remoe it from the tree and later inserting it to a new location.

Note: *node removal without removing its childs is out of the scope of this article. For now, we assume, that*
node to remove has no children, i.e. right-left=1

and by removing the original node.

```
var nodetoremove = db.categoriesNSO.findOne({_id:"LG"});

if((nodetoremove.right-nodetoremove.left-1)>0.001) {
    print("Only node without childs can be removed")
    exit
}

var followingsibling = db.categoriesNSO.findOne({_id:"Cell_Phones_and_Accessori

//update all remaining nodes
db.categoriesNSO.update({right:{$gt:nodetoremove.right}},{$inc:{right:-2}}, fal
db.categoriesNSO.update({left:{$gt:nodetoremove.right}},{$inc:{left:-2}}, false
db.categoriesNSO.remove({_id:"LG"});
```

Let's check the result:

```
+-Electronics (1,44)
  +--Cameras_and_Photography (2,13)
        +-----Digital_Cameras (3,4)
        +-----Camcorders (5,6)
        +-----Lenses_and_Filters (7,8)
        +-----Tripods_and_supports (9,10)
        +-----Lighting_and_studio (11,12)
    +---Shop_Top_Products (14,23)
        +-----IPad (15,16)
        +-----IPhone (17,18)
        +-----IPod (19,20)
        +-----Blackberry (21,22)
    +---Cell_Phones_and_Accessories (24,43)
        +-----Cell_Phones_and_Smartphones (25,36)
              +--------Nokia (26,27)
              +--------Samsung (28,29)
              +--------Apple (30,31)
              +--------HTC (32,33)
              +--------Vyacheslav (34,35)
          +------Headsets (37,38)
          +------Batteries (39,40)
          +------Cables_And_Adapters (41,42)
```

Updating/Moving a Single Node

You can move the node within the same parent, or to another parent. If you're moving it within the same parent and the nodes are without childs, then you just need to exchange the node's **left** and **right** pairs.

The formal way of doing this is to remove the node and insert it to a new destination, thus the same restrictions apply - only nodes without children can be moved.

If you need to move a subtree, consider creating a mirror of the existing parent under a new location, and move the nodes under the new parent one

As an example, let's move the `LG` node from the insertion example under the `Cell_Phones_and_Smartphones` node as its last sibling (i.e. you do not have following sibling node as in the insertion example)

Step 1 would be to remove the `LG` node from the tree using the node removal procedure described above.

Step2 is to take the right value of the new parent.

The new node will have the left value of its parent's right value. The right value will be its parent's right value incremented by 1.

Now we have to create the place for the new node: update affects right values of all nodes on a further traversal path

```
var newparent = db.categoriesNSO.findOne({_id:"Cell_Phones_and_Smartphones"});
var nodetomove = {_id:'LG', left:newparent.right,right:newparent.right+1}


//3th and 4th parameters: false stands for upsert=false and true stands for mul
db.categoriesNSO.update({right:{$gte:newparent.right}},{$inc:{right:2}}, false,
db.categoriesNSO.update({left:{$gte:newparent.right}},{$inc:{left:2}}, false, t

db.categoriesNSO.insert(nodetomove)
```

Let's check the result:

```
+-Electronics (1,46)
  +--Cameras_and_Photography (2,13)
        +-----Digital_Cameras (3,4)
        +-----Camcorders (5,6)
        +-----Lenses_and_Filters (7,8)
        +-----Tripods_and_supports (9,10)
        +-----Lighting_and_studio (11,12)
    +---Shop_Top_Products (14,23)
        +-----iPad (15,16)
        +-----iPhone (17,18)
        +-----iPod (19,20)
        +-----Blackberry (21,22)
    +---Cell_Phones_and_Accessories (24,45)
        +-----Cell_Phones_and_Smartphones (25,38)
                +---------Nokia (26,27)
                +---------Samsung (28,29)
                +---------Apple (30,31)
                +---------HTC (32,33)
                +---------Vyacheslav (34,35)
                +---------LG (36,37)
            +-------Headsets (39,40)
            +-------Batteries (41,42)
            +-------Cables_And_Adapters (43,44)
```

Getting the Node Children (unordered)

Note: unless all node childs have no children themselves, it is impossible to get a node's direct child.

Consider using the modified approach of combining `NestedSets` with the parent field.

Getting all Node Descendants

This is the core strength of this approach - all descendants will be retrieved using one select to DB. Moreover, if you

```
var descendants=[]
var item = db.categoriesNSO.findOne({_id:"Cell_Phones_and_Accessories"});
print ('('+item.left+','+item.right+')')
var children = db.categoriesNSO.find({left:{$gt:item.left}, right:{$lt:item.rig
while(true === children.hasNext()) {
  var child = children.next();
  descendants.push(child._id);
}


descendants.join(",")
//Cell_Phones_and_Smartphones,Headsets,Batteries,Cables_And_Adapters,Nokia,Sams
```

Getting the Path to Node

Retrieving the path to node is also elegant in this approach, and it can be done with a single query to the database

```
var path=[]
var item = db.categoriesNSO.findOne({_id:"Nokia"})

var ancestors = db.categoriesNSO.find({left:{$lt:item.left}, right:{$gt:item.ri
while(true === ancestors.hasNext()) {
  var child = ancestors.next();
  path.push(child._id);
}

path.join('/')
// Electronics/Cell_Phones_and_Accessories/Cell_Phones_and_Smartphones
```

Indexes

```
db.categoriesAAO.ensureIndex( { left: 1, right:1 } )
```

Tree Structure using a Combination

of Nested Sets and Classic Parent Reference with order approach

For each node, we'll store the **ID**, **Parent**, **Order**, **left**, and **right**:

ID	Parent	Order	Left	Right
Cell_Phones_and_Smartphones	Cell_Phones_and_Accessories	10	25	36
HTC	Cell_Phones_and_Smartphones	40	32	33

The left field is also treated as an order field, so we can omit an order field. However, we also can leave it. This way we can use the Parent Reference with the order data to reconstruct the left/right values in the case of accidental corruption, or, for example during the initial import.

Adding a New Node

Adding a new node can be adopted from Nested Sets in this manner:

```
var followingsibling = db.categoriesNSO.findOne({_id:"Cell_Phones_and_Accessori
var previoussignling = db.categoriesNSO.findOne({_id:"Shop_Top_Products"});
var neworder = parseInt((followingsibling.order + previoussignling.order)/2);
var newnode = {_id:'LG', left:followingsibling.left,right:followingsibling.left
db.categoriesNSO.update({right:{$gt:followingsibling.right}},{$inc:{right:2}},
db.categoriesNSO.update({left:{$gte:followingsibling.left}, right:{$lte:followi

db.categoriesNSO.insert(newnode)
```

```
        +----Cameras_and_Photography (2,13)  ord.[10]
        +-----Shop_Top_Products (14,23)  ord.[20]
        +-----Cell_Phones_and_Accessories (26,45)  ord.[30]
```

After insertion:

```
+--Electronics (1,46)
        +----Cameras_and_Photography (2,13)  ord.[10]
                    +-------Digital_Cameras (3,4)  ord.[10]
                    +-------Camcorders (5,6)  ord.[20]
                    +-------Lenses_and_Filters (7,8)  ord.[30]
                    +-------Tripods_and_supports (9,10)  ord.[40]
                    +-------Lighting_and_studio (11,12)  ord.[50]
        +-----Shop_Top_Products (14,23)  ord.[20]
                    +-------IPad (15,16)  ord.[10]
                    +-------IPhone (17,18)  ord.[20]
                    +-------IPod (19,20)  ord.[30]
                    +-------Blackberry (21,22)  ord.[40]
        +-----LG (24,25)  ord.[25]
        +-----Cell_Phones_and_Accessories (26,45)  ord.[30]
                    +-------Cell_Phones_and_Smartphones (27,38)  ord.[10]
                                +----------Nokia (28,29)  ord.[10]
                                +----------Samsung (30,31)  ord.[20]
                                +----------Apple (32,33)  ord.[30]
                                +----------HTC (34,35)  ord.[40]
                                +----------Vyacheslav (36,37)  ord.[50]
                    +--------Headsets (39,40)  ord.[20]
                    +--------Batteries (41,42)  ord.[30]
                    +--------Cables_And_Adapters (43,44)  ord.[40]
```

Updating/Moving a Single Node

This will be identical to the insertion approach

This will use the approach from Nested Sets.

Getting the Node Children (ordered)

This is finally possible by using the (Parent,Order) pair

```
db.categoriesNSO.find({parent:"Electronics"}).sort({order:1});
/*

{ "_id" : "Cameras_and_Photography", "parent" : "Electronics", "order" : 10, "l
{ "_id" : "Shop_Top_Products", "parent" : "Electronics", "order" : 20, "left" :
{ "_id" : "LG", "left" : 24, "right" : 25, "parent" : "Electronics", "order" :
{ "_id" : "Cell_Phones_and_Accessories", "parent" : "Electronics", "order" : 30

*/
```

Getting all Node Descendants

You can use the Nested Set's approach for this.

Getting the Path to Node

You can use the Nested Set's approach for this.

The Code in Action

You can download the code from Github.

All files are packaged according to the following naming convention:

- **MODELReference.js** - initialization file with tree data for MODEL approach
- **MODELReference_operating.js** - add/update/move/remove/get children

- **MODELReference_pathtonode.js** - code illustrating how to obtain path to node
- **MODELReference_nodedescendants.js** - code illustrating how to retrieve all the descendants of a node

Final Words

Please note that MongoDB does not provide ACID transactions. This means that for update operations split
into separate update commands, your application should implement additional code to support your code-specific transactions.

MongoDB's formal advice is as follows:

- The Parent Reference pattern provides a simple solution to tree storage, but requires multiple queries to retrieve subtrees
- The Child References pattern provides a suitable solution to tree storage as long as no operations on subtrees are necessary. This pattern may also provide a suitable solution for storing graphs where a node may have multiple parents.
- The Array of Ancestors pattern has no specific advantages unless you constantly need to get the path to node

You are free to mix patterns (by introducing an order field, etc) to match the data operations required to your application.

Node.js MongoDB DevOps Database

3 12 SHARE

Vyacheslav

Experienced Software Engineer, with Project Management Experience

High load web projects, bespoke software development, project management experience DevOps: Ansible,Vagrant, Chef PAAS & Cloud: (AWS, Amazon beanstalk, Redhat openshift, OpenStack, Digital Ocean) Continuous integration (with Jenki...

FOLLOW

12 Replies

Leave a reply

Yong Z a month ago

Google “Sorted-Unlimited-Depth-Tree” or go here: https://github.com/drinkjava2/Multiple-Columns-Tree, a new solution simpler than nested sets. I’ve confirmed with Joe Celko (author of 《Trees and Hierarchies in SQL》) it’s a new solution.

Reply

Vyacheslav a month ago

Rule 1: Do not spam

Rule 2: If you comment, make sure you are commenting on topic.

Question: what it has to do with mongo?

Enjoy this post?

3

Yong Z a month ago

Your article is talking store tree structure in mongo, and one of them is “Nested Sets”, I just give a better solution than “Nested Sets”. What’s the relationship between “Nested Sets” and Mongo? no, “Nested Sets” is a algorithm invented before than Mongo. I did not give a mongo implementation example (not my job) doesn’t mean the new algorithm can not be implemented on mongo, in fact it’s pretty simple.

Reply

Vyacheslav a month ago

So why would you not post your own article, and stop polluting/spamming google output with your “better solutions” in comments? :) or alternatively, implement it in mongo, than your comment will be more or less correlating to topic.

Reply

Yong Z a month ago

Algorithm is more important than implementation, why you are so worry about “related to Mongo”? OK, I give up, here is the mongo implement to show the new solution:

Books

Show more

Reply

Vyacheslav a month ago

Crazy 8)

Vyacheslav a year ago

@Keizer - under operations under subtrees I mean need of retrieving all subnodes at ones, like in e-shop - get all the subcategories of the parent one.

Generally, the less queries to database are invoked on most popular operations in your application - the better is.

♡ 1 Reply

Keizer a year ago

Helpfull post, thanks!

> *The Child References pattern provides a suitable solution to tree storage as long as no operations on subtrees are necessary.*

Show more

♡ Reply

Show more replies

Joan Manuel Vasquez

A simple CRUD using MySQL and Node Js

CRUD is an acronym for "CREATE, READ, UPDATE, DELETE" in SQL. Basically what we are going to demonstrate is how to use MySQL in Node JS. There are lots of tutorials out there on how to make a CRUD using Mongo DB, but what we really want to show in here is how to use SQL in Node Js.

WHAT DO YOU NEED TO FOLLOW THIS TUTORIAL?

ES6/ES7

MYSQL

SOME BASIC KNOWLEDGE OF NODE JS

WHAT ARE WE GOING TO BUILD?

We are going to create a TODO LIST with Node JS using ES6 and Common JS. All we need in this project is "promise-mysql". Let's install: npm i -s promise-mysql

READ MORE

Enjoy this post?

♡ 3