ELASTICSEARCH JOINS: HAS_CHILD, HAS_PARENT QUERY

Once again we tackle the complexity and sometimes contradictory documentation of ElasticSearch and try to make it easier to understand. Here we look at how to parent-child relationships between documents.

The Concepts Parent, Child, and Join

In a relational database a parent-child relationship is called a **join**. A mathematician would call that the intersection of two sets. For example, we can find the **intersection** of a set odd numbers and prime numbers, creating a set of odd numbers that are not prime.

A database person would express that in a parent-child relationship. To do that it is required are some common element.

To illustrate, suppose we have these two sets some **odd numbers** and **some prime numbers**. Then we make the intersection of the two sets, i.e., the set of elements that are in both. A database join is the same thing.

```
parent (some odd numbers)                    {9,21}
children (some prime number)                 {7,11, 21}
intersection (odd numbers n prime numbers) {21}
```

It does not matter in this case which set we pick as parent or child as these are just numbers.

We will illustrate this in ElasticSearch using universities and students. We will create a set of universities and students and then assign students to each.

So we will have something like:

parent (universities)	{Harvard, Clemson}
children (students)	{{"student:" Walker", "university": "Clemson}, {student: Stephen", "university": "Harvard"}}
intersection (students who go to Clemson)	{"student:" Walker", "university": "Clemson}

The child points to the parent because the university name is in the chile record. That is the common element. In terms of ElasticSearch you indicate this in the index mapping like this:

```
"universities_students" : {
          "type" : "join",
            "eager_global_ordinals" : true,
            "relations" : {
              "universities" : "students"
          }
     }
```

This creates a new field with the name **universities_students** that is of type **join**. The parent is **universities** and the child is **students**.

When you write a student (child) record you point it to its parent using the **_id**, i.e., **'parent'**: '**e2cfb3b015d4be50a466aa593ab4d9f490982ede**' of the parent document, and indicate that it is a document of type **students**:

```
'universities_students': {
'name': 'students',
 'parent': 'e2cfb3b015d4be50a466aa593ab4d9f490982ede'
}
```

And when you write a university (parent) record you indicate that it is a document of type **universities**. You put no _id as that would not be logical. The query figures out that relationship.

```
'universities_students': {
'name': 'universities'
}
```

ElasticSearch is Not a Database

ElasticSearch points out that it is **not** a relational database. So it does not let you do traditional joins because those would run too slow, and ES is all about speed. So they require that both document types must be in the same shard on the same index. That puts them close together on the same disk drive, thus reducing seek time. They also point out this query will run slower as you add more documents even with that rule.

First create mapping Type

Now, we illustrate this with an example. First create a mapping type. Note that this mapping type will be able to contain both universities and students. This is because this list of fields is the superset of all fields common to both. That is, students have **classes** and **grades**, but universities do not. Similarly schools have an **AdminEmail**, school name etc.

Notice that at the end we list the field **universities_students** that is of type **join**. This expresses the relationship between documents.

(We will only put the school name in the universities record. You can ignore all of those other fields, like phone number. We used them in this previous example on how to do nested queries, which is a related, albeit different topic.)

```
curl -XPUT --header 'Content-Type: application/json'
http://parisx:9200/universities -d '{
"settings" : {
        "index" : {
            "number_of_shards" : 1,
            "number_of_replicas" : 1
        }
    },    "mappings" : {
      "doc" : {
        "properties" : {
          "Address" : { "type" : "text"},
          "AdminEmail" : { "type" : "text"},
          "AdminName" : { "type" : "text" },
          "AdminPhone" : { "type" : "text" },
          "DapipId" : { "type" : "text" },
          "Fax" : { "type" : "text" },
          "GeneralPhone" : { "type" : "text" },
          "LocationName" : { "type" : "text" },
          "LocationType" : { "type" : "text" },
          "OpeId" : { "type" : "text" },
          "ParentDapipId" : { "type" : "text" },
          "ParentName" :  { "type" : "text" },
          "UpdateDate" : { "type" : "text" },
          "classes" : {
                "type" : "nested",
                "properties" : {
                   "name" : { "type" : "text"},
                   "grades" : { "type" : "integer" }
              }
            },
        "firstName" : { "type" : "text" },
        "lastName" : { "type" : "text" },
        "school" : { "type" : "text" },
          "universities_students" : {
```

```
          "type" : "join",
            "eager_global_ordinals" : true,
            "relations" : {
              "universities" : "students"
          }
          }
        }
        }
}
}'
```

Create Data

Download and run this code. To make things simple we create only one school and one student. If you want to play around with this and create lots more data just change the **range(1,2)** to something like **range(1,400)**.

You could create data using curl, as in the ES examples in their documentation. But it's much easier to use Python, since it's easier to work with Python dictionaries than JSON objects, particularly when the ES documents are long and have lots of objects within objects,

Look at the university and student records below. Note that when you run it it will generate different data since it picks schools at random. You need to note the **_id** of the university record so that you can plug it into the query below which finds children records.

parent(university)

```
{'school': 'University of South Carolina - Columbia',
'universities_students': {'name': 'universities'}}
{'_version': 1, '_type': 'doc', '_id':
'e2cfb3b015d4be50a466aa593ab4d9f490982ede', '_primary_term': 1, 'result':
'created', '_index': 'universities', '_seq_no': 0, '_shards': {'failed': 0,
'total': 2, 'successful': 2}}
```

child (students)

```
{'school': 'University of North Carolina at Chapel Hill',
'universities_students': {'name': 'students', 'parent':
'e2cfb3b015d4be50a466aa593ab4d9f490982ede'}, 'lastName': 'Mann', 'classes': ,
'firstName': 'Walker'}
```

Here is the complete code. Below that we explain those parts which are not self-evident. It just picks one of three schools based on a random number. It picks names and classes using this same approach. Then it adds the university and student record as we explain above.

The basic approach is to create a dictionary {} then add fields to it. We can add dictionaries to dictionaries to create nested JSON objects. So it's simpler than working with long JSON string fields.

```
import requests
import random
import hashlib
import json

url="http://parisx:9200/universities/doc/"

def assignClasses():
    classes=[]
    for x in range(random.randrange(1,6)):
      classes.append(genclass())
    return classes
schools = ("Arizona State University", "University of South Carolina -
Columbia", "University of North Carolina at Chapel Hill")
firstNames = ("Walker", "Stephen", "Julie", "George")
lastNames = ("Rowe", "Shakespeare", "Mann", "Sarte")
courses= ("math", "physics", "French", "logic")

def genclass():
    classes={}
    classes=courses
    classes = random.randrange(1,7)
    return classes

def genParent(id):
    child={}
    child = 'students'
    child = id
    return child

def genSchool():
    school={}
    school = {'name': 'universities'}
    school=schools
    m = hashlib.sha1()
    m.update(bytes(json.dumps(school), 'utf-8'))
    id = m.hexdigest()
    print(school)
    response = requests.post(url + id, json=school)
    print (response.json())
    return id

def genStudent():
    id = genSchool()
    students={}
    students = genParent(id)
    students=schools
```

```
    students = firstNames
    students = lastNames
    students = assignClasses()
    m = hashlib.sha1()
    m.update(bytes(json.dumps(students), 'utf-8'))
    sid = m.hexdigest()

    print(students)
    response = requests.post(url + sid + "?routing=1&refresh", json=students)
    print (response.json())

for r in range(1,2):
    genStudent()
```

```
url="http://parisx:9200
/universities/doc/"
```

The index/type remains one of the most confusing parts of ElasticSearch. They say they have dropped the **document type** field, but when you write data you are requested to use it in the **index/type**. It's acontradiction. In this case that is **doc** in **universities/doc**.

```
m = hashlib.sha1()
m.update(bytes(json.
dumps
(school),'utf-8'))
id = m.hexdigest()
```

Every ElasticSearch document must have a unique ID. It you write the same document ID twice one will erase the other. The common approach is to calculate a SHA1 digest over the whole JSON document. Of course two documents could have the same ID. You could also use something like uuid.UUID.

```
response = requests.post
(url + sid
+ "?routing=1&refresh",
json=students)
```

When we add the child document we are required to tell it what shard to use since it must be on the same shard and the parent document. (A shard is a division of an index. It is designed to distribute data across nodes in a cluster. This is not a limit. We can still have more than one shard. Would just need to put the routing command on the parent as well and use the same shard number as we rotate through a list of those.) Put told ES how many shards to create when we created the index map:

```
"index" : {
            "number_of_shards" : 1,
            "number_of_replicas" : 1
        }
```

Find children

Now have have create some data. So we can run some queries.

The first finds children records given a parent. The key word here is **parent_id**. The type of object we are looking for is "type": "students". Ad we tell it what school to look for by putting the _id of the school in **"id": "e2cfb3b015d4be50a466aa593ab4d9f490982ede"**.

```
curl -XGET --header 'Content-Type: application/json'
http://parisx:9200/universities/_search?pretty -d '{
```

```
  "query": {
    "parent_id": {
      "type": "students",
      "id": "e2cfb3b015d4be50a466aa593ab4d9f490982ede"
    }
  }
}'
```

This query lists this student (remember we only added one student.):

```
{
        "_index" : "universities",
        "_type" : "doc",
        "_id" : "185fa8736085fd07ce6f5fb8f1c66fce64fa42e3",
        "_score" : 0.18232156,
        "_routing" : "1",
        "_source" : {
          "classes" : ,
          "school" : "University of South Carolina - Columbia",
          "lastName" : "Mann",
          "universities_students" : {
            "name" : "students",
            "parent" : "e2cfb3b015d4be50a466aa593ab4d9f490982ede"
          },
          "firstName" : "Walker"
        }
      }
```

Find Parents

There is no field in the parent which points to the children, so we cannot just put the children id into the query. That would not be logical as if we knew the children IDs then why bother referring to the parent at all. Instead we can list all parent docs that have a child.

We use the **exists** query parameter since we are required to put something in the query parameter.

```
curl -XGET --header 'Content-Type: application/json'
http://parisx:9200/universities/_search?pretty -d '{
    "query": {
        "has_child" : {
            "type" : "students",
            "query" : {
                  "exists" : {"field": "firstName"}
        }
    }
}
}'
```

This results in the school (parent) record being listed:

```
      {
       "_index" : "universities",
       "_type" : "doc",
       "_id" : "e2cfb3b015d4be50a466aa593ab4d9f490982ede",
       "_score" : 1.0,
       "_source" : {
         "school" : "University of South Carolina - Columbia",
         "universities_students" : {
           "name" : "universities"
         }
}
```