Chapter 4: Patterns (Part 2)

Lab: Tree Patterns

Back to the Question

Correct Option:

- Parent References

```
{
  "_id": "<objectId>",
  "name": "<string>",
  "role": "<string>",
  "department": {
    "name": "<string>",
    "id": "<objectId>"
  },
  "reports_to": { "id": "<objectId>", "name": "<string>" }
}
```

Using the **Parent References** pattern would be most efficient and straightforward approach given the requirements.

Using as an example *"Jon Yullin"* and his manager *"Stuart Spencer"*, we would get the following:

```
{
  "_id": ObjectId("123414123"),
  "name": "Jon Yullin",
  "role": "Intern",
  "department": {
```

```
    "name": "Engineering",
    "id": ObjectId("988576342364")
  },
  "reports_to": { "id": ObjectId("123414123") , "name":
"Stuart Spencer" },
}
```

One single database request to find the direct manager of a given employee

COPY

```
> db.employees.find({"name": "Jon Yullin"})
{
  "_id": ObjectId("123414123"),
  "name": "Jon Yullin",
  "role": "Intern",
  "department": {
    "name": "Engineering",
    "id": ObjectId("988576342364")
  },
  "reports_to": { "id": ObjectId("123414124") , "name":
"Stuart Spencer" }
}
```

In this schema, the *reports_to* field gives the application the direct manager information.

Collect all direct reports of an employee with one single and efficient query

COPY

```
> db.employees.createIndex({"reports_to.name": 1})
> db.employees.find({"reports_to.name": "Stuart Spencer"})
{
  "_id": ObjectId("123414123"),
  "name": "Jon Yullin",
  "role": "Intern",
  "department": {
    "name": "Engineering",
    "id": ObjectId("988576342364")
  },
  "reports_to": { "id": ObjectId("123414124") , "name":
"Stuart Spencer" }
}
```

In this example, querying all of our employees for anyone with the value of *"Stuart Spencer"* for the name of their manager, *reports_to.name* will give us a list of all direct reports.

Creating a single field index on *reports_to.name* makes this query efficient.

One single update operation to change the reporting structure of an employee

```
> db.employees.updateOne(
  {"name": "Jon Yullin"},
  {"$set": {"reports_to": { "id":  ObjectId("123414122"),
"name": "Jalpa Maganin" }}}
)
{
  "_id": ObjectId("123414123"),
  "name": "Jon Yullin",
  "role": "Intern",
  "department": {
    "name": "Engineering",
    "id": ObjectId("988576342364")
  },
  "reports_to": { "id":  ObjectId("123414122"), "name":
"Jalpa Maganin" }
}
```

Or in the case we would like replace a manager to a set of direct

Incorrect Options:

- Array of Ancestors

```
{
  "_id": "<objectId>",
  "name": "<string>",
  "role": "<string>",
  "department": {
    "name": "<string>",
    "id": "<objectId>"
  },
  "reports_to": [ { "id": "<objectId>", "name": "<string>" }
]
}
```

Array of Ancestors is a pattern that allows us to perform all of the requested queries in an efficient way, however the update operation would not be possible with a single update operation.

One single update operation to change the reporting structure of an employee

To exemplify this scenario let's promote *"Stuart Spencer"* and have him report to *"Jalpa Maganin"*

The *reports_to* field in "Stuart Spencer" document could be performed with a single update:

```
> var new_reporting_structure = [
  { "id": ObjectId("123414120") , "name": "Maria Gutierrez"
},
  { "id": ObjectId("123414123") , "name": "Jalpa Maganin" }
  ]
> db.employees.updateOne(
    {"name": "Stuart Spencer"},
    {"$set": {"reports_to": new_reporting_structure, "role":
"Manager"}}
  )
{
  "_id": ObjectId("123414123"),
  "name": "Stuart Spencer",
  "role": "Manager",
  "department": {
    "name": "Engineering",
    "id": ObjectId("988576342364")
  },
  "reports_to": [
    { "id": ObjectId("123414120") , "name": "Maria Gutierrez"
},
    { "id": ObjectId("123414123") , "name": "Jalpa Maganin"
},
  ]
}
```

However, we would also need to run several updates, to reflect this new structure, on all of "Stuart Spencer" subsequent reports, which requires at least one other update.

That said, the *Array of Ancestors* is often the right solution when you need to find all direct and indirect reports.

- Materialized Paths

```
{
  "_id": "<objectId>",
```

```
  "name": "<string>",
  "role": "<string>",
  "department": {
    "name": "<string>",
    "id": "<objectId>"
  },
  "reports_to": "<string>/<string>/<string>"
}
```

Using *"Jon Yullin"* example we would get:

COPY

```
{
  "_id": ObjectId("123414123"),
  "name": "Jon Yullin",
  "role": "Intern",
  "department": {
    "name": "Engineering",
    "id": ObjectId("988576342364")
  },
  "reports_to": "Maria Gutierrez/Jalpa Maganin/Kimi
Macha/Stuart Spencer" }
}
```

The Materialized Paths approach falls short on:

Collect all direct reports of an employee with one single and **efficient** *query*

We can use a regular expression to match all reports of a given employee. However, this regular expression query, to be efficient and use a supporting index, requires the prepending of left end side of the *reports_to* values. In this particular case, if we do not have the information of the root of the organization chart, the CEO name, we would not be able to create an efficient query.

COPY

```
db.employees.createIndex({"reports_to": 1})
// does not use the index on `reports_to`
db.employees.find({"reports_to": /Stuart\ Spencer/)
// can use the index on `reports_to` field
db.employees.find({"reports_to": /^Maria\ Gutierrez.*Stuart\
Spencer/)
```

Proceed to next section