Learning NoSQL — NoSQL Database Designing

This article covers the most important topic in NoSQL as how to efficiently structure collections/documents.

Danish Siddiq · Follow

3 min read · Sep 24, 2019

912 6

In part-I, SQL vs NoSQL a basic concept was introduced along with the MongoDB fundamentals. This article covers the most controversial, debatable and important topic in NoSQL.

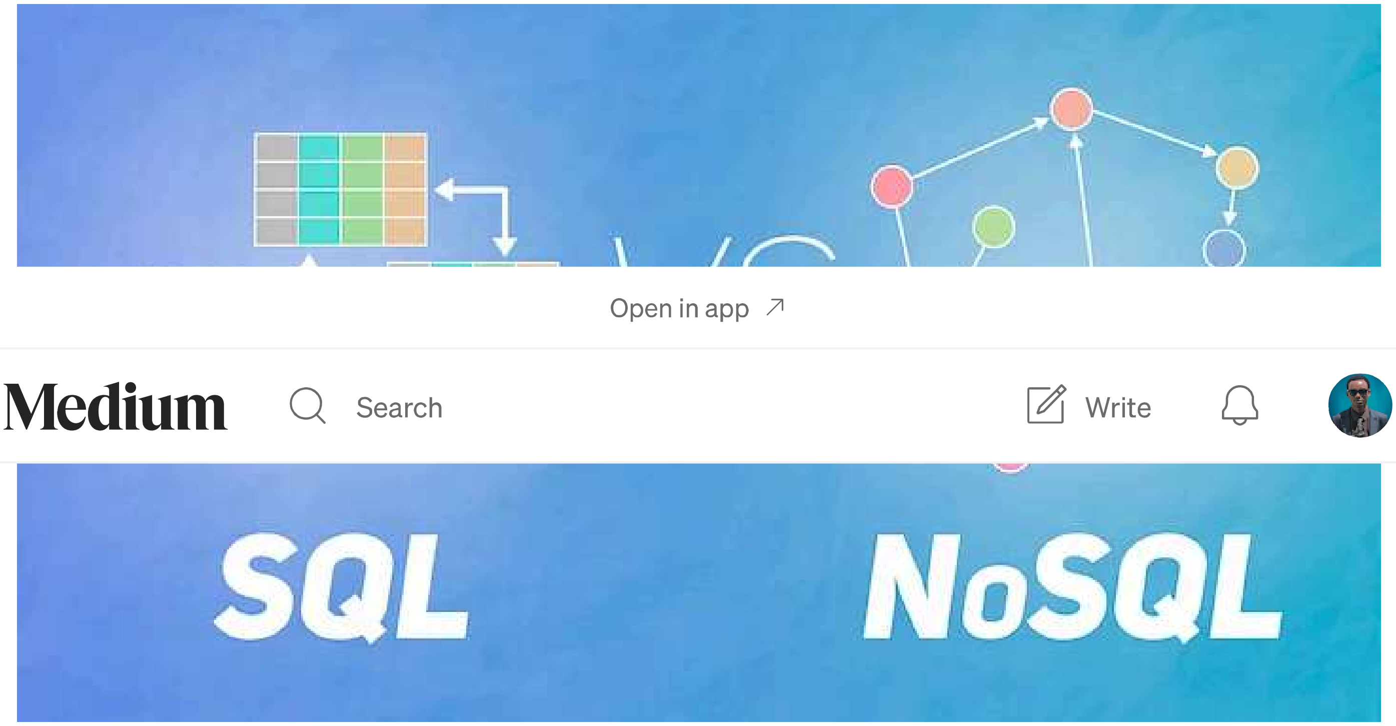


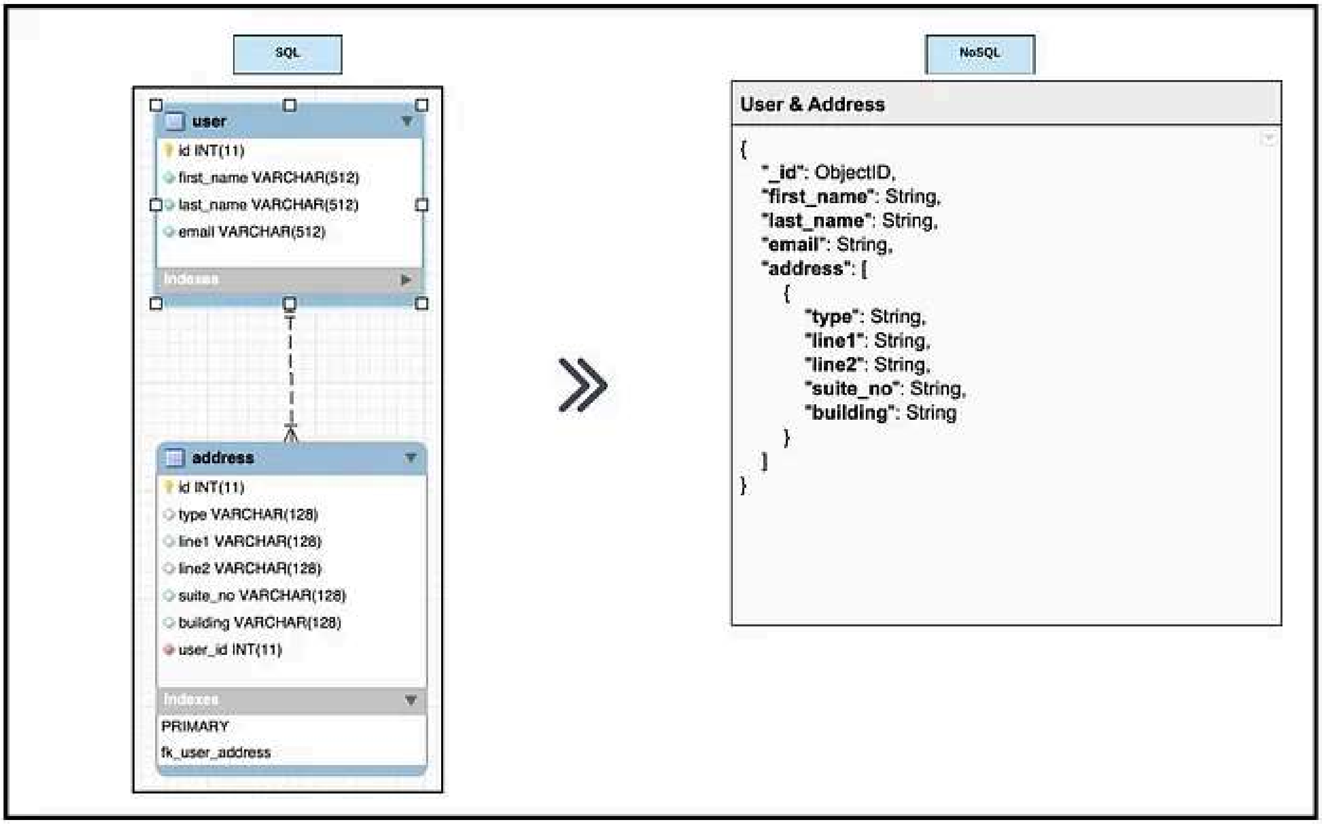
Table vs Collection

Modeling in NoSQL vs SQL:

I will try to explain a concept with multiple examples. Modeling scenarios may differ in situations depending on requirements. In modeling a schema, it must be noted that MongoDB manages documents with a maximum size of 16MB.

In NoSQL, either define a collection of nested objects or multiple collections where each contains a simple object definition. In the following few cases, we discuss SQL to NoSQL schema transformation.

Case A: User Address:



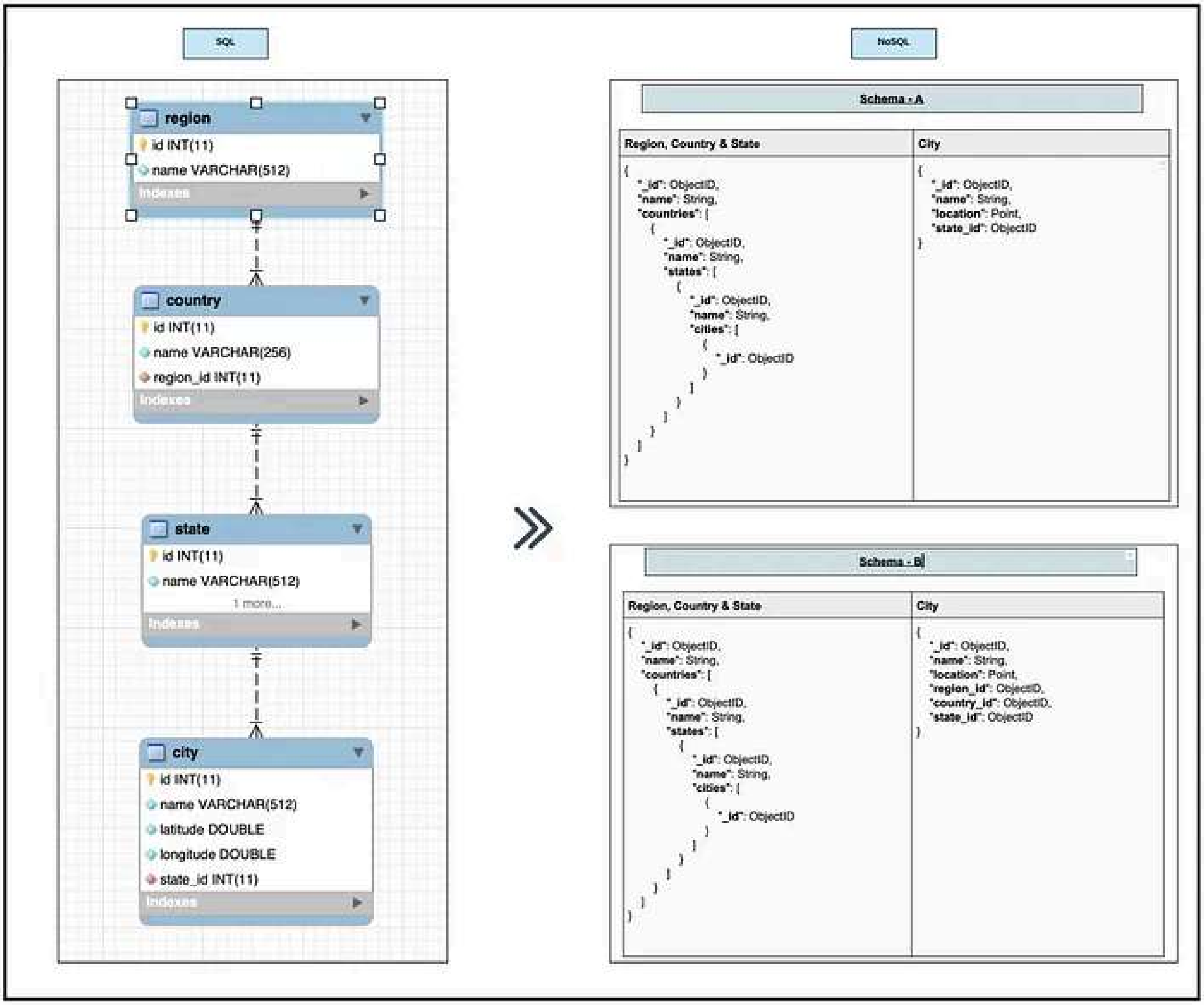
```
{
  "_id": ObjectID,
  "first_name": String,
  "last_name": String,
  "email": String,
  "address": [
    {
      "type": String,
      "line1": String,
      "line2": String,
      "suite_no": String,
      "building": String
    }
  ]
}
```

Simple 1:M transformation from SQL to NoSQL

Design Explanation:

1. The address has been moved as an array is flexible enough to accommodate additional entries in the future.
2. Each item in the address array has a limited number of fields, no need to create a separate collection.

Case B: Region, Country, State, and City Model — Approach I:



Schema - A

Region, Country & State

```
{
  "_id": ObjectID,
  "name": String,
  "countries": [
    {
      "_id": ObjectID,
      "name": String,
      "states": [
        {
          "_id": ObjectID,
          "name": String,
          "cities": [
            {
              "_id": ObjectID
            }
          ]
        }
      ]
    }
  ]
}
```

City

```
{
  "_id": ObjectID,
  "name": String,
  "location": Point,
  "state_id": ObjectID
}
```

Schema - B

Region, Country & State

```
{
  "_id": ObjectID,
  "name": String,
  "countries": [
    {
      "_id": ObjectID,
      "name": String,
      "states": [
        {
          "_id": ObjectID,
          "name": String,
          "cities": [
            {
              "_id": ObjectID
            }
          ]
        }
      ]
    }
  ]
}
```

City

```
{
  "_id": ObjectID,
  "name": String,
  "location": Point,
  "region_id": ObjectID,
  "country_id": ObjectID,
  "state_id": ObjectID
}
```

Multiple dependent 1:M relationships transformation from SQL to NoSQL

Design Explanation:

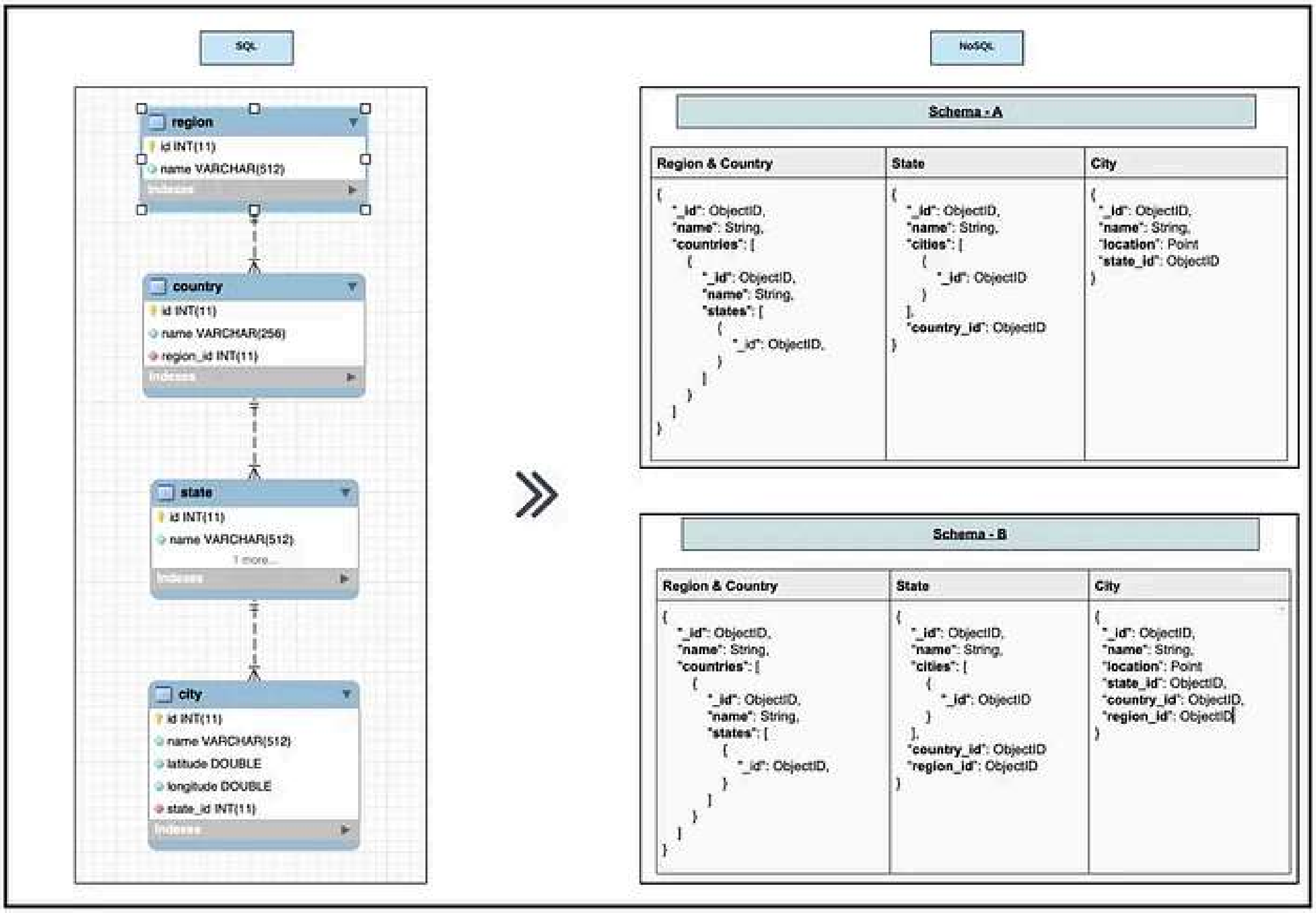
1. 1-M relationships are advised to handle in a single model where data size does not increase drastically. Therefore region, country, and state are handled in a single document which is kind of a defined set of data.
2. Cities are kept in a separate collection for 2 reasons:

- For some countries, The number of cities is huge and this will make document size much bigger.
- To avoid region unnecessary nested and complex modeling.

Schema - A vs B — Better Query Performance:

1. The city collection provides wider range of options to query data from region collection efficiently and faster, based on “region_id”, “country_id” and “state_id”.
2. Cities can be grouped easily based on “region_id”, “country_id” or “state_id” more efficiently.

Case C: Region, Country, State, and City Model — Approach II:



Schema - A

Region & Country

```
{
   "_id": ObjectID,
   "name": String,
   "countries": [
      {
         "_id": ObjectID,
         "name": String,
         "states": [
            {
               "_id": ObjectID,
            }
         ]
      }
   ]
}
```

State

```
{
   "_id": ObjectID,
   "name": String,
   "cities": [
      {
         "_id": ObjectID
      }
   ],
   "country_id": ObjectID
}
```

City

```
{
 "_id": ObjectID,
 "name": String,
 "location": Point
 "state_id": ObjectID
}
```

Schema - B

Region & Country

```
{
   "_id": ObjectID,
   "name": String,
   "countries": [
      {
         "_id": ObjectID,
         "name": String,
         "states": [
            {
               "_id": ObjectID,
            }
         ]
      }
   ]
}
```

State

```
{
   "_id": ObjectID,
   "name": String,
   "cities": [
      {
         "_id": ObjectID
      }
   ],
  "country_id": ObjectID
  "region_id": ObjectID
}
```

City

```
{
 "_id": ObjectID,
 "name": String,
 "location": Point
 "state_id": ObjectID,
 "country_id": ObjectID,
 "region_id": ObjectID
}
```

A perfect example of multiple dependent 1:M relationship transformation from SQL to NoSQL

Design Explanation:

1. This schema has less nested objects therefor more clarity

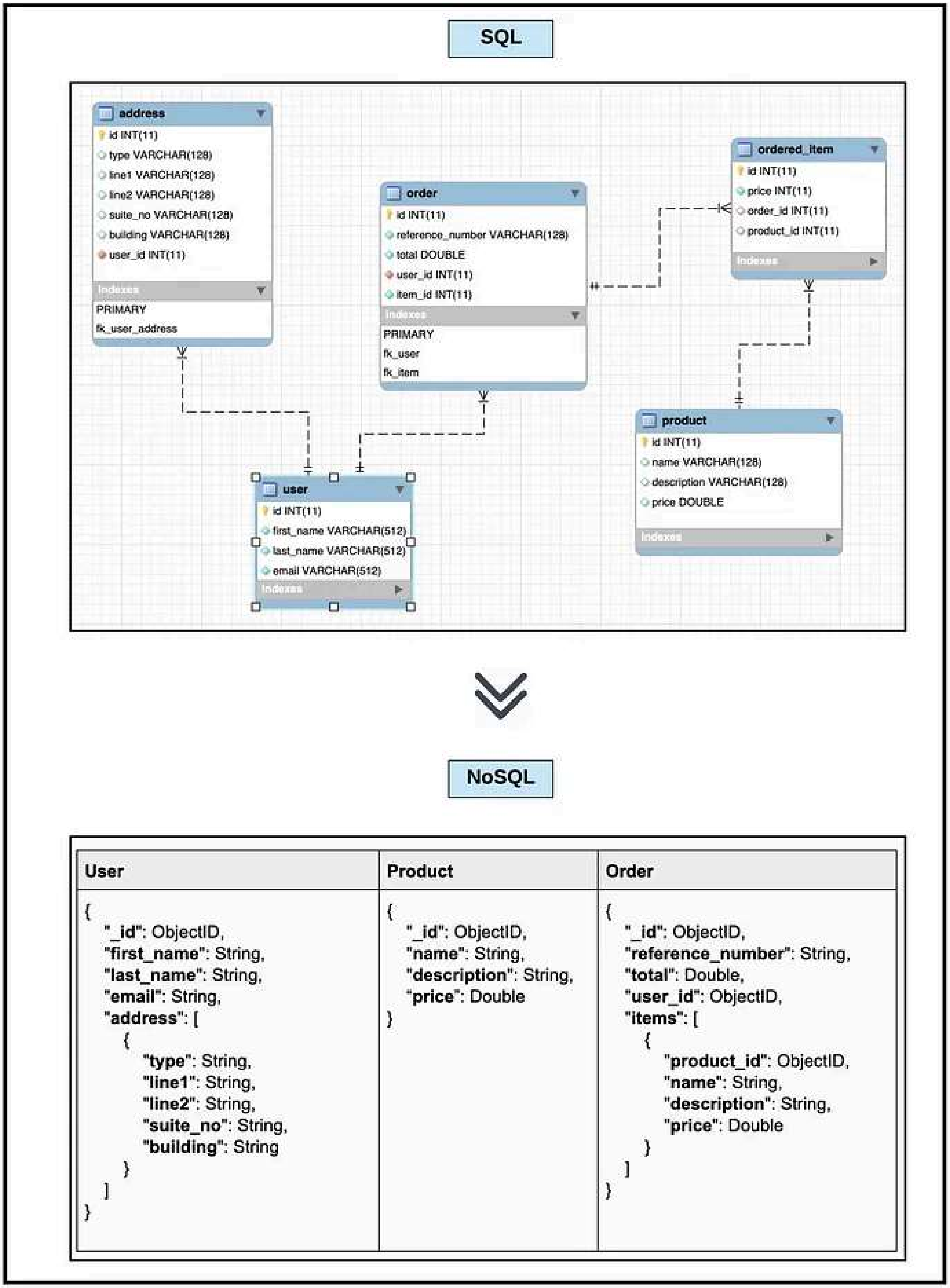
2. It is much easier for applications such as NodeJS apps to handle queries based on predefined schemas.

Schema — A vs B — Better Query Performance:

1. B does contain redundancy but in NoSQL, performance is preferred to normalization.
2. The data in the above example is relatively limited, but consider a business case where millions of documents are involved, the performance of Schema-A is no match for Schema-B throughput.

Case Online Users, Product and Orders:

Country and city references are ommited in the following address schema for the purpose of simplicity, focusing on M:M relationship:



User	Product	Order

{
"**_id**": ObjectID,
"**first_name**": String,
"**last_name**": String,
"**email**": String,
"**address**": [
{
"**type**": String,
"**line1**": String,
"**line2**": String,
"**suite_no**": String,
"**building**": String
}
]
}
	{
	"**_id**": ObjectID,
	"**name**": String,
	"**description**": String,
	"**price**": Double
	}
		{
		"**_id**": ObjectID,
		"**reference_number**": String,
		"**total**": Double,
		"**user_id**": ObjectID,
		"**items**": [
		{
		"**product_id**": ObjectID,
		"**name**": String,
		"**description**": String,
		"**price**": Double
		}
		]
		}

Combination of 1:M(user-address) & 1:M(user-order) and M:M(order-product) relationships

Design Explanation:

1. M:M relationship of order and product is handled only by one collection.

Conclusion:

1. Do not over complicate a structure by extensive nested objects
2. (1:M) & (M:M) relationships are easier to transform into a single collection if document sizes are reasonable.
3. Split nested objects into separate collections, either for simplicity or size
4. Normalization is not a priority in NoSQL. Its power comes with its flexibility. NoSQL is the best option for unstructured data, its focus is on how to respond with quick throughput.

Upcoming:

Next article focuses on the MongoDB and few tools installation to start exploring about MongoDB in detail.

MongoDB installation and configuration — Part III

In part II of this series, modeling in NoSQL was covered in detail with multiple examples. This article will cover…

medium.com

NoSQL Database Mongodb Sql Nodejs