

Modelado de Datos en MongoDB

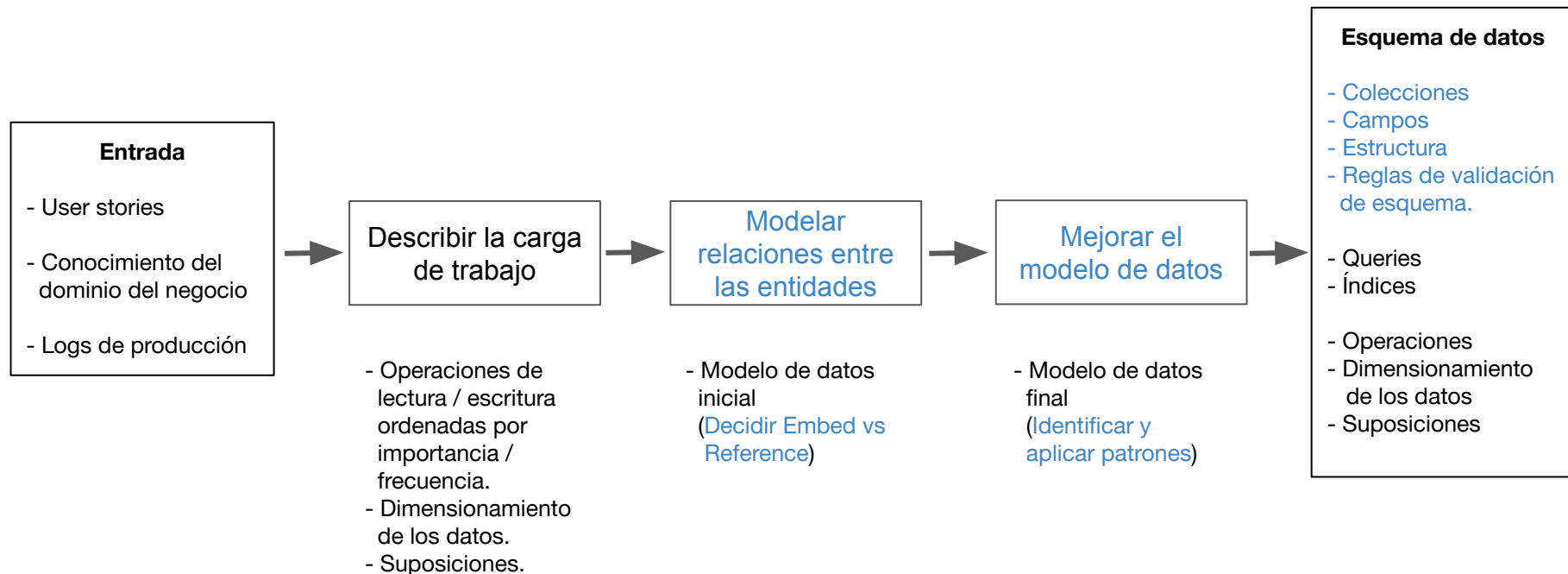
Bases de Datos NoSQL

Bases de Datos, 2023

Modelado de Datos en MongoDB

- Esquema de Datos Flexibles.
 - Proporciona capacidad para adaptarse a las necesidades cambiantes.
 - No hay que definir esquemas fijos para las colecciones.
 - Validación de Esquemas.
 - En la práctica los datos tienen algún tipo de “estructura fijos”.
 - A partir de la versión 3.6, MongoDB provee validación de esquemas.
 - Encontrar un balance entre flexibilidad y validación de esquemas.
- Diseñar un buen modelo de datos.
 - ¿Para qué?
 - Garantizar una buena performance.
 - Minimizar el costo general de la solución de la aplicación.
 - ¿Cómo?
 - No hay una fórmula, pero...
 - Existe un metodología.

Metodología



Validación de esquemas en MongoDB

- MongoDB es capaz de realizar validación de esquemas durante la actualización o inserción de documentos
- Las reglas de validación se definen por cada colección
- Se pueden definir al crear una colección:
 - `db.createCollection(name, options)` con la opción `validator`
- Se pueden definir sobre una colección ya creada, con el comando `collMod`:

- `db.runCommand({ collMod: <collection or view>, validator: <value>, ... })`

Validación de esquemas en MongoDB

- **Ejemplo:**

```
> db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name" ],
      properties: {
        name: {
          bsonType: "string",
          description: "must be a string and is required"
        },
        year: {
          bsonType: "int",
          minimum: 1900,
          maximum: 3000,
          description: "must be an integer in [ 1900, 3000 ] and is not required"
        }
      }
    }
  }
})
```

Validación de esquemas en MongoDB

- **Ejemplos:**

```
> db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name" ],
      properties: {
        name: {
          bsonType: "string",
        },
        year: {
          bsonType: "int",
          minimum: 1900,
          maximum: 3000,
        }
      }
    }
  }
})
```

```
> db.students.insert({"name": "Mary", "year": "2020"})
```



Validación de esquemas en MongoDB

- **Ejemplos:**

```
> db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name" ],
      properties: {
        name: {
          bsonType: "string",
        },
        year: {
          bsonType: "int",
          minimum: 1900,
          maximum: 3000,
        }
      }
    }
  }
})
```

```
> db.students.insert({"name": "Mary", "year": "2020"})
```



Document failed validation ("year" no es int)

```
> db.students.insert({
  "name": "Mary", "year": NumberInt(2020)
})
```



Validación de esquemas en MongoDB

- Ejemplos:

```
> db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name" ],
      properties: {
        name: {
          bsonType: "string",
        },
        year: {
          bsonType: "int",
          minimum: 1900,
          maximum: 3000,
        }
      }
    }
  }
})
```

```
> db.students.insert({"name": "Mary", "year": "2020"})
```



Document failed validation ("year" no es int)

```
> db.students.insert({
  "name": "Mary", "year": NumberInt(2020)
})
```



El shell de mongo trata todos los numeros como **double** por default. Es necesario usar el constructor NumberInt

```
> db.students.insert({ "name": "Mary" })
```



Validación de esquemas en MongoDB

- Ejemplos:

```
> db.createCollection("students", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      required: [ "name" ],
      properties: {
        name: {
          bsonType: "string",
        },
        year: {
          bsonType: "int",
          minimum: 1900,
          maximum: 3000,
        }
      }
    }
  }
})
```

```
> db.students.insert({"name": "Mary", "year": "2020"})
```



Document failed validation (“year” no es int)

```
> db.students.insert({
  "name": "Mary", "year": NumberInt(2020)
})
```



El shell de mongo trata todos los numeros como **double** por default. Es necesario usar el constructor NumberInt

```
> db.students.insert({ "name": "Mary" })
```



“year” no es un campo requerido

Validación de esquemas en MongoDB

Ver metadata de las colecciones

- De todas las colecciones: **db.getCollectionInfos()**
- De una colección específica: `db.getCollectionInfos({"name": "students"})`
- Ejemplo:

```
> db.getCollectionInfos({"name": "students"})
[
  {
    "name" : "students",
    "type" : "collection",
    "options" : {
      "validator" : {
        "$jsonSchema" : {
          "bsonType" : "object",
          "required" : [ "name" ],
          ...
        }
      }
    }
  }
]
```

Validación de esquemas en MongoDB

Documentos existentes

- Cuando se agrega validación a una colección, los documentos existentes no son validados hasta que son modificados.
- La opción `validationLevel` determina en qué operaciones se aplican las reglas de validación.
- Si `validationLevel` es `strict` (default), MongoDB aplica las reglas de validación en todos los inserts y updates.
- Si `validationLevel` es `moderate`, MongoDB aplica las reglas de validación en inserts y updates de documentos existentes que ya cumplan los criterios de validación.

Validación de esquemas en MongoDB

Ejemplo:

```
> db.contacts.insertMany([
  {
    "_id": 1,
    "name": "Anne",
    "phone": "123456",
    "city": "London",
    "status": "Complete"
  },
  {
    "_id": 2,
    "name": "Ivan",
    "city": "Vancouver"
  }
])
```

Si se actualiza el documento con:

- `_id: 1`, se valida, dado que el documento cumple los criterios de validación
- `_id: 2`, no se valida, dado que no posee un campo "phone"

```
> db.runCommand( { // modificación de una colección existente
  collMod: "contacts",
  validator: { $jsonSchema: {
    bsonType: "object",
    required: [ "phone", "name" ],
    properties: {
      phone: {
        bsonType: "string",
        description: "must be a string and is required"
      },
      name: {
        bsonType: "string",
        description: "must be a string and is required"
      }
    }
  } },
  validationLevel: "moderate"
} )
```

Validación de esquemas en MongoDB

Documentos inválidos

- La opción `validationAction` determina cómo MongoDB maneja los documentos que no cumplen las reglas de validación.
- Si `validationAction` toma el valor `error` (*default*), MongoDB rechaza cualquier inserción o actualización que no cumpla las reglas de validación.
- Si `validationAction` toma el valor `warn`, MongoDB advierte que hubo una violación a la reglas de validación, pero permite la inserción o actualización.

JSON Schema

- **JSON Schema** es la manera recomendada para realizar validación de schema en MongoDB
- El operador `$jsonSchema` matchea documentos que satisfagan el JSON Schema especificado
- Sintaxis:
 - `{ $jsonSchema : <JSON Schema object> }`
- Donde `<JSON Schema object>` debe ser formateado de acuerdo al [draft 4 of the JSON Schema standard](https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/#op._S_jsonSchema):
 - `{ <keyword1>: <value1>, ... }`

JSON Schema

- Además de su finalidad de validación, MongoDB permite utilizar el operador `$jsonSchema` en operaciones de lectura/escritura donde una “query” sea requerida, para encontrar los documentos en la colección que satisfagan el esquema especificado.
- Ejemplos:

```
> db.collection.find( { $jsonSchema: <schema> } )  
> db.collection.aggregate( [ { $match: { $jsonSchema: <schema> } } ] )  
> db.collection.updateMany( { $jsonSchema: <schema> }, <update> )  
> db.collection.deleteOne( { $jsonSchema: <schema> } )
```

- Para encontrar documentos que no satisfagan el schema, usar `$nor`:

```
> db.collection.find( { $nor: [ { $jsonSchema: <schema> } ] } )  
  
...
```

JSON Schema - Keywords

Listado de Keywords principales

- **bsonType**, enumera los posibles tipos aceptados. Acepta los mismos [string aliases](#) que el operador [\\$type](#).
- **required**, solo para objetos. Especifica los campos que el objeto debe contener.
- **properties**, solo para objetos. Un JSON Schema, donde cada uno de sus valores es también un JSON Schema.

JSON Schema - Keywords

Ejemplos:

```
{
  $jsonSchema: {
    required: [ "name", "major", "gpa", "address" ],
    properties: {
      name: {
        bsonType: "string",
      },
      address: {
        bsonType: "object",
        required: [ "zipcode" ],
        properties: {
          "street": { bsonType: "string" },
          "zipcode": { bsonType: "string" }
        }
      }
    }
  }
}
```

```
{
  $jsonSchema: {
    bsonType: "object",
    required: [ "name", "birthdate" ],
    properties: {
      name: {
        bsonType: "string",
      },
      birthdate: {
        bsonType: [ "string", "date" ],
      }
    }
  }
}
```

JSON Schema - Keywords

Listado de Keywords principales

- **enum**, enumera todos los valores posibles del campo.
- **maximum, minimum** solo para campos de tipo numérico.
- **maxLength, minLength** solo para campos de tipo string.
- **maxItems, minItems** solo para campos de tipo array.
- **description**, un string que describe el schema y no tiene efecto en la validación.

<https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/#available-keywords>

JSON Schema - Keywords

Ejemplo:

```
$jsonSchema: {
  required: [ "year", "major" ],
  properties: {
    year: {
      bsonType: "int",
      minimum: 2017,
      maximum: 3017,
      description: "must be an integer in [ 2017, 3017 ] and is required"
    },
    major: {
      enum: [ "Math", "English", "Computer Science", "History", null ],
      description: "can only be one of the enum values and is required"
    },
  },
}
```

JSON Schema - Keywords

Ejemplo:

```
$jsonSchema: {
  required: [ "year", "major" ],
  properties: {
    year: {
      bsonType: "int",
      minimum: 2017,
      maximum: 3017,
    },
    major: {
      enum: [ "Math", "English",
              "Computer Science",
              "History", null ],
    },
  }
}
```

```
> db.students.insertOne(
  {"year": NumberInt(1900), "major": "Math"}
)
```



JSON Schema - Keywords

Ejemplo:

```
$jsonSchema: {
  required: [ "year", "major" ],
  properties: {
    year: {
      bsonType: "int",
      minimum: 2017,
      maximum: 3017,
    },
    major: {
      enum: [ "Math", "English",
              "Computer Science",
              "History", null ],
    },
  },
}
```

```
> db.students.insertOne(
  {"year": NumberInt(1900), "major": "Math"}
)
```



“year” no es mayor a 2017

```
> db.students.insertOne(
  {"year": NumberInt(2020), "major": "foo"}
)
```



JSON Schema - Keywords

Ejemplo:

```
$jsonSchema: {
  required: [ "year", "major" ],
  properties: {
    year: {
      bsonType: "int",
      minimum: 2017,
      maximum: 3017,
    },
    major: {
      enum: [ "Math", "English",
              "Computer Science",
              "History", null ],
    },
  }
}
```

```
> db.students.insertOne(
  {"year": NumberInt(1900), "major": "Math"}
)
```



“year” no es mayor a 2017

```
> db.students.insertOne(
  {"year": NumberInt(2020), "major": "foo"}
)
```



“major” no corresponde a ningun valor del enum

```
> db.students.insertOne(
  {"year": NumberInt(2020), "major": "Math"}
)
```



JSON Schema - Keywords

Ejemplo:

```
$jsonSchema: {
  required: [ "year", "major" ],
  properties: {
    year: {
      bsonType: "int",
      minimum: 2017,
      maximum: 3017,
    },
    major: {
      enum: [ "Math", "English",
              "Computer Science",
              "History", null ],
    },
  }
}
```

```
> db.students.insertOne(
  {"year": NumberInt(1900), "major": "Math"}
)
```



“year” no es mayor a 2017

```
> db.students.insertOne(
  {"year": NumberInt(2020), "major": "foo"}
)
```



“major” no corresponde a ningun valor del enum

```
> db.students.insertOne(
  {"year": NumberInt(2020), "major": "Math"}
)
```



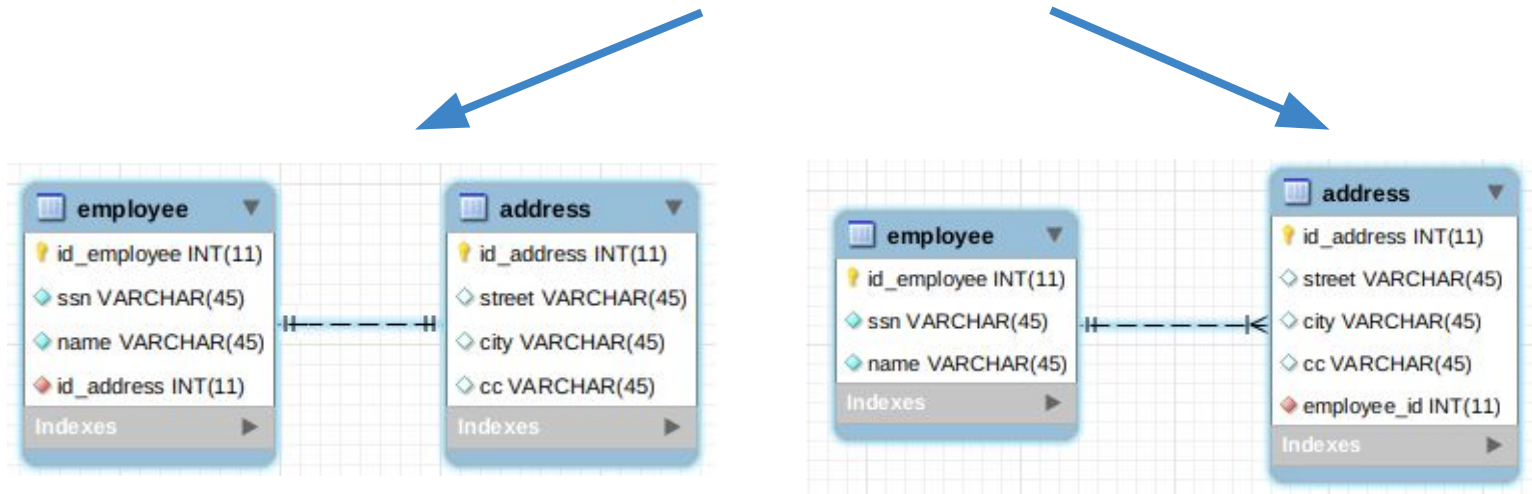
JSON Schema - Keywords

Listado de Keywords principales

- Para conocer el listado completo de keywords disponibles en MongoDB:
<https://docs.mongodb.com/manual/reference/operator/query/jsonSchema/#available-keywords>
- Keywords extras que pueden aparecer en el práctico o en el parcial:
 - **items**
 - **uniqueItems**
 - **pattern**
 - **additionalProperties**
- Material sugerido para profundizar sobre keywords:
<https://json-schema.org/understanding-json-schema/>

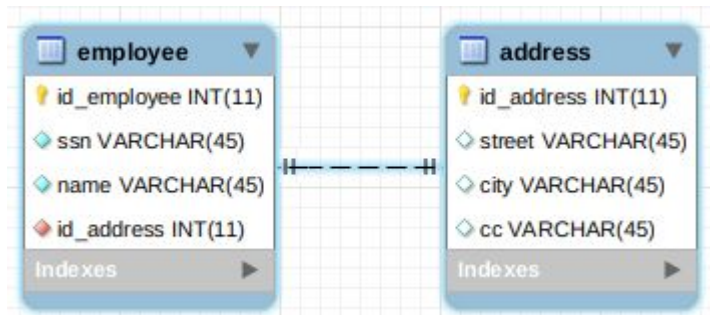
SQL a MongoDB: Modelar Relaciones

- ¿Cómo modelar relaciones “one-to-one” o “one-to-N” en MongoDB?



Modelar Relaciones: One-to-One

- Patrón Embedding



“address” es embebido dentro del documento de la colección “employee”

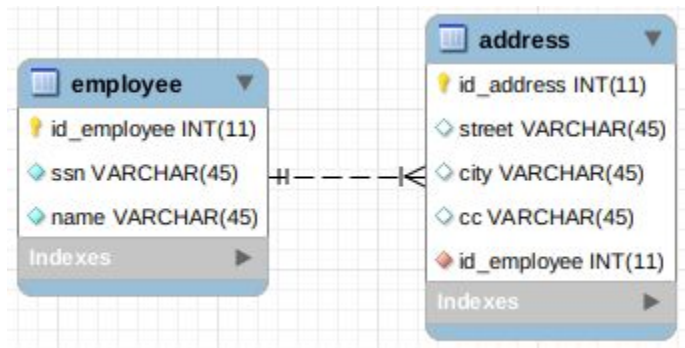
```
db.employee.insert(  
  {  
    "_id": "employee1",  
    "name": "Kate Monster",  
    "ssn": "123-456-7890",  
    "address" : {  
      "street": "123 Sesame St",  
      "city": "Anytown",  
      "cc": "USA"  
    }  
  }  
)
```

Modelar Relaciones: One-to-N

- Criterios y restricciones a tener en cuenta al modelar una relación one-to-N
 - El tamaño máximo de un documento BSON es de 16 MB.
 - ¿La entidad del lado N, de una relación One-to-N, tiene sentido que esté fuera del contexto del documento padre?
- La relación One-to-N se puede descomponer en 3 casos básicos
 - Si $2 \leq N < 100$ entonces considero aplicar el modelado **One-to-Few**.
 - Si $100 \leq N < 1000$ entonces considero aplicar el modelado **One-to-Many**.
 - Si $N \geq 1000$ entonces considero aplicar el modelado **One-to-Squillon**.

Modelar Relaciones: One-to-Few

- Patrón Embedding

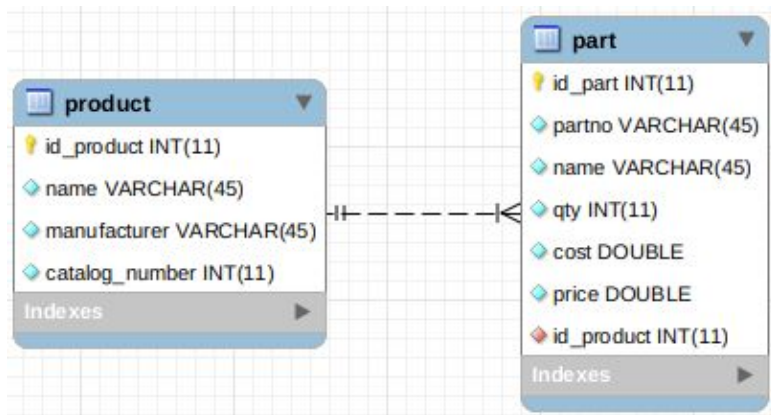


“addresses” son embebidas en un arreglo dentro del documento de la colección “employee”

```
db.employee.insert(  
  {  
    "_id": "employee1",  
    "name": "Kate Monster",  
    "ssn": "123-456-7890",  
    "addresses" : [  
      {  
        "street": "123 Sesame St",  
        "city": "Anytown",  
        "cc": "USA"  
      },  
      {  
        "street": "123 Avenue Q",  
        "city": "New York",  
        "cc": "USA"  
      }  
    ]  
  }  
)
```

Modelar Relaciones: One-to-Many

- Patrón Child-Referencing



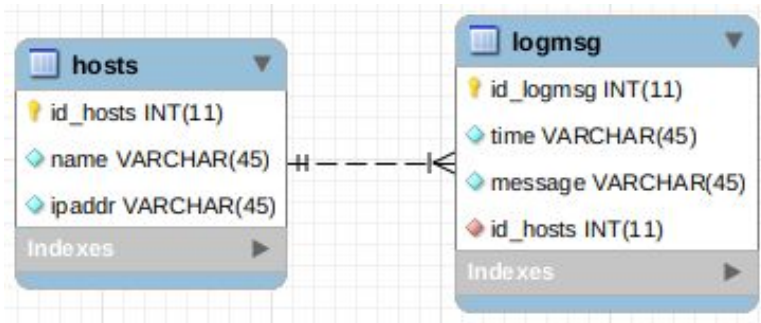
Arreglo de referencias de los `_id` de los documentos de la colección "part"

```
db.part.insert([
  {
    "_id": "part1", partno: "123-aff-456",
    name: "#4 grommet", qty: 94,
    cost: 0.94, price: 3.99
  },
  {
    "_id": "partN", partno: "123-aff-678",
    name: "#5 grommet", qty: 94,
    cost: 0.98, price: 3.29
  }
])
```

```
db.product.insert({
  "_id": "product1",
  name: "left-handed smoke shifter",
  manufacturer: "Acme Corp",
  catalog_number: 1234,
  parts: [ "part1", "partN" ]
})
```

Modelar Relaciones: One-to-Squillon

- Patrón Parent-Referencing



Referencia a un documento de la colección "hosts"

```
db.hosts.insert({
  _id : "host1",
  name : "goofy.example.com",
  ipaddr : "127.66.66.66"
})

db.logmsg.insert([
  {
    _id: 1000001,
    time : ISODate("2014-03-28T09:42:41"),
    message : "cpu is on fire!",
    id_host: "host1"
  },
  {
    _id: 1000002,
    time : ISODate("2014-03-28T09:49:41"),
    message : "cpu is iddle!",
    id_host: "host1"
  }
])
```

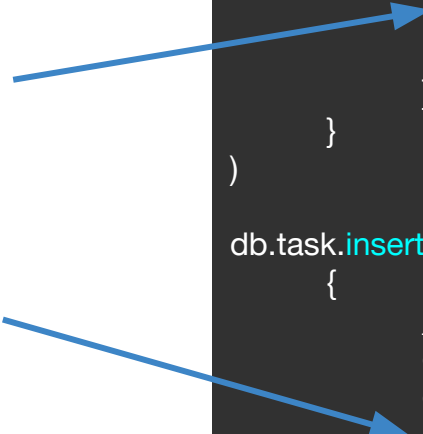
Modelar Relaciones: One-to-Many

- Patrón Two-Way Referencing

Arreglo de referencias a los documentos de la colección "task"

Referencia al documento de la colección "employee"

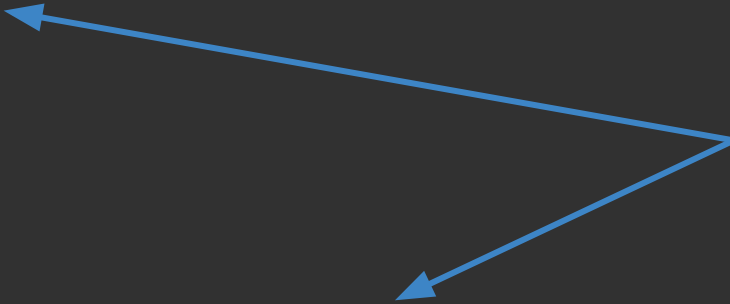
```
db.employee.insert(  
  {  
    "_id": "employee1",  
    "name": "Kate Monster",  
    "tasks" : [  
      "task1",  
      "taskN"  
    ]  
  }  
)  
  
db.task.insert([  
  {  
    _id: "task1",  
    description: "Write lesson plan",  
    due_date: ISODate("2018-04-04"),  
    owner: "employee1"  
  }  
)
```



Modelar Relaciones: One-to-Many con desnormalización

```
db.part.insert([
  { "_id": "part1", partno: "123-aff-456", name: "#4 grommet", qty: 94, cost: 0.94, price: 3.99 },
  { "_id": "partN", partno: "123-aff-678", name: "#5 grommet", qty: 94, cost: 0.98, price: 3.29 }
])

db.product.insert({
  "_id": "product1",
  name: "left-handed smoke shifter",
  manufacturer: "Acme Corp",
  catalog_number: 1234,
  parts: [ { part_id: "part1", name: "#4 grommet" }, { part_id: "partN", name: "#5 grommet" } ]
})
```



Agrega redundancia
para mayor eficiencia
en la consulta

- Se deben considerar los siguientes factores:
 - No se puede realizar un update atómico en datos desnormalizados.
 - La desnormalización solo tiene sentido cuando se tiene alta relación lectura / escritura.

Resumen de reglas para modelar relaciones

- Favorecer “embedding” a menos que haya una razón convincente para no hacerlo.
- La necesidad de acceder a la entidad por sí sola es una razón para no usar “embedding”.
- Arreglos que crecen sin límite es una razón para no usar “embedding”.
- Considere la relación lectura / escritura al desnormalizar.
- La forma en que se modela los datos en MongoDB depende, en su totalidad, de los patrones de accesos a los datos de la aplicación en particular.

Patrones de Diseño

- ¿Qué es un patrón de diseño?
 - Describe la solución a un problema de diseño recurrente.
 - No es una solución completa.
 - Facilita la reutilización de la solución.
- Patrones que vamos a ver.
 - Schema Versioning.
 - Computed.
 - Polymorphic.

- Catálogo de Patrones

		Use Case Categories						
		Catalog	Content Management	Internet of Things	Mobile	Personalization	Real-Time Analytics	Single View
Patterns	Approximation	✓		✓	✓		✓	
	Attribute	✓	✓					✓
	Bucket			✓			✓	
	Computed	✓		✓	✓	✓	✓	✓
	Extended Reference	✓			✓		✓	
	Outlier			✓	✓	✓		
	Preallocated			✓			✓	
	Polymorphic	✓	✓		✓			✓
	Schema Versioning	✓	✓	✓	✓	✓	✓	✓
	Subset	✓	✓		✓	✓		
	Tree and Graph	✓	✓					

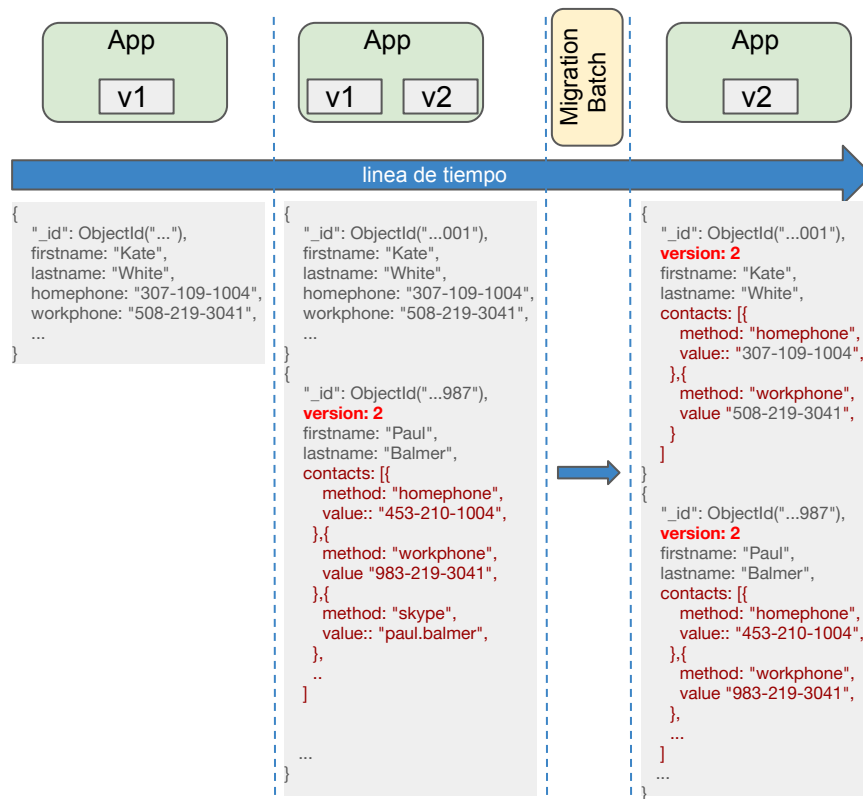
Patrones de Diseño: Schema Versioning

- Problema

- Evitar periodo de inactividad mientras se actualiza el esquema de datos.
- Actualizar todos los documentos puede demorar horas, días o aun semanas cuando se trata de big data.
- No queremos actualizar todos los docs

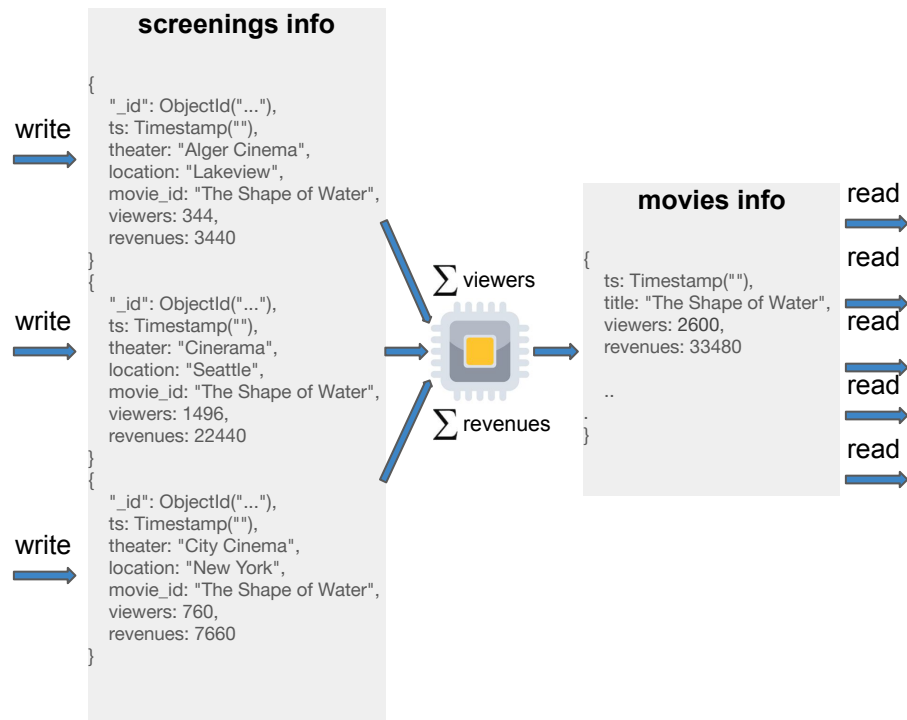
- Solución

- A cada documento nuevo se le agrega un campo **version**.
- La aplicación puede manejar todas las versiones.
- Elegir la estrategia para migrar los documentos a la nueva versión.



Patrones de Diseño: Computed

- Problema
 - Existen datos que necesitan ser procesados con frecuencia.
 - El acceso a los datos está dominado por lecturas (ej, 1M reads / 1K writes).
- Solución
 - Realizar el procesamiento y almacenar el resultado en la colección apropiada.
 - Si se necesita rehacer el cálculo, mantener el origen de los datos.
- Ventajas
 - Operaciones de lecturas más rápidas.
 - Ahorro de recursos (CPU y storage).



Patrones de Diseño: Polymorphic

- Problema

- Los documentos tienen más similitudes que diferencias. Y queremos,
- Mantenerlos en la misma colección.

- Solución

- Especificar el o los campos que rastreen el tipo de (sub) documento.
- La aplicación debe tener fragmentos de códigos distintos por cada tipo de documento o bien tener subclases.

- Ventajas

- Sencillo de implementar.
- Permite realizar consultas en una sola colección.

- Campos en común

<pre>{ product_type: "shirt", size: "large", price: 100.00, color: "blue" }</pre>	<pre>{ product_type: "book", size: "20 cm x 15 cm x 1 cm", price: 10.00, title: "Learning Serverless" }</pre>	campos en común
---	---	-----------------

- Polimorfismo en subdocumentos

<pre>{ sport: "tennis", athlete_name: "Martina Navratilova", events: [{ type: "singles", career_tournaments: 390, career_titles: 167 }, { type: "doubles", career_tournaments: 233, career_titles: 177, partners: ["Tomanova", "Fernandez", "Morozova", "Mezert", "..."] }] ... }</pre>	campos en común
	campos en común

Referencias

- Sitio oficial de JSON Schema: <https://json-schema.org/>
- Understanding JSON Schema: <https://json-schema.org/understanding-json-schema/>
- Modelado de relaciones en MongoDB
<https://docs.mongodb.com/manual/applications/data-models-relationships/>
- 6 rules of thumb para modelado de relaciones en MongoDB
<https://www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1>
- Building with Patterns: A Summary.
<https://www.mongodb.com/blog/post/building-with-patterns-a-summary>