CHAPTER 4

Schema Modelling

In databases, the *schema* defines the internal structure or organization of the data. In relational databases like MySQL or Postgres, the schema is implemented as tables and columns.

MongoDB is often described as a schema-free database, but this is somewhat misleading. By default, MongoDB does not enforce any particular document structure, but all MongoDB applications will implement some sort of document model. It's therefore more accurate to describe MongoDB as supporting *flexible* schemas.

In MongoDB, a schema is implemented by the collections – which generally represent sets of similar documents – and the structure of the documents within those collections.

The performance limits of a MongoDB application are largely determined by the document model that the application implements. The amount of work that an application needs to do to retrieve or process information is primarily dependent on how that information is distributed across multiple documents. In addition, the size of documents will determine how many documents MongoDB can cache in memory. These and many other trade-offs will determine how much physical work the database will have to do to satisfy a database request.

Although MongoDB does not have the equivalent of the expensive and time-consuming SQL `ALTER TABLE` statement, it remains very difficult to make fundamental changes to a document model once it has been established and deployed in production. Choosing the correct data model is, therefore, a critical early task in the design of your application.

You could fill up a book on the topic of data modelling, and indeed some have. In this chapter, we'll try to cover the core tenants of data modelling from a performance perspective.

The Guiding Principles

Ironically, schema modelling with MongoDB flexible schemas can actually be harder than in the fixed schemas of the relational database.

In relational database modelling, you model the data logically, eliminating redundancy until you achieve the *third normal form*. Simplistically, third normal form is achieved when every element in a row is dependent on the key, the whole key and nothing but the key.[1] You then introduce redundancy through *denormalization* to support performance objectives. The resulting data model usually remains roughly in third normal form but with some slight modifications to support critical queries.

You could model MongoDB documents into third normal form, but it would almost always be the wrong solution. MongoDB is designed around the idea that you should include almost all relevant information within a single document – not spread it across multiple entities as you would in the relational model. Therefore, instead of creating a model based on the structure of the data, you create a model based on the structure of your queries and updates.

Here are the key objectives of MongoDB data modelling:

- **Avoid joins**: MongoDB supports a simple join capability using the aggregation framework (see Chapter 7). However, in contrast to a relational database, joins are expected to be an exception, not the rule. Aggregation-based joins are unwieldy, and it's more typical for data to be joined within the application code. In general, we try to ensure that our critical queries can find all the data they need within a single collection.
- **Manage redundancy**: By encapsulating relevant data into a single document, we create a problem of redundancy – we may have more than one place in the database where a certain data element can be found. For instance, consider a `products` collection and an `orders` collection. The `orders` collection will probably include product names within the order details. If we need to change a product name, we'll have to change it in multiple places. This will make that update operation potentially very time-consuming.

[1]In honor of the creator of the relational model Edgar Codd, we would often say "the key, the whole key and nothing but the key, so help me Codd!"

- **Beware of the 16MB limit**: MongoDB has a 16MB limit on the size of an individual document. We need to make sure that we never try to embed so much information that we risk exceeding that limit.
- **Maintain consistency**: MongoDB does support transactions (see Chapter 9), but they require special programming and have significant constraints. If we want to atomically update sets of information, it can be advantageous to include those data elements in a single document.
- **Monitor memory**: We want to ensure that most operations on MongoDB documents occur in memory. However, if we make our documents very large by embedding lots of information, then we reduce the number of documents that can fit in memory and might increase IO. Therefore, we want to keep documents small when we can.

Linking vs. Embedding

There are a wide variety of MongoDB schema design patterns, but they all involve variations of these two approaches:

- *Embedding* everything in a single document.
- *Linking* collections using pointers to data in other collections. This is roughly equivalent to using a relational database's third normal form model.

A Case Study

There's much room for compromise between the linking and embedding approaches and a lot of non-performance-related reasons for choosing one over the other (atomic updates and the 16M document limit, for instance). Nevertheless, let's look at how the two extremes compare from a performance point of view – at least for a specific workload.

For this case study, we will model the classic "Orders" schema. An Orders schema includes orders, details about the customer that created the order, and the products that comprise the order. In a relational database, we'd diagram this schema as in Figure 4-1.

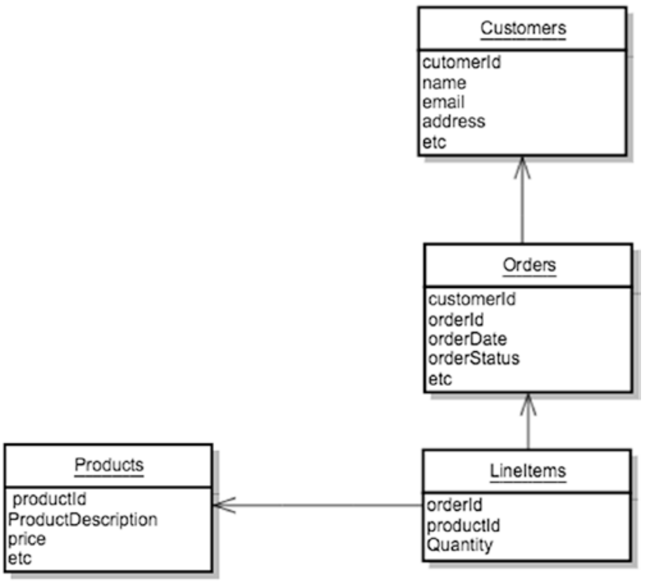


Figure 4-1. *Orders-products schema in relational form*

If we were to model this schema using only the linking paradigm, we would create a collection for each of the four logical entities. They might look something like this:

```
mongo>db.customers.findOne();
{
        "_id" : 3,
        "first_name" : "Danyette",
        "last_name" : "Flahy",
        "email" : "dflahy2@networksolutions.com",
        "Street" : "70845 Sullivan Center",
        "City" : "Torrance",
        "DOB" : ISODate("1967-09-28T04:42:22Z")
}
mongo>db.orders.findOne();
```

```
{
      "_id" : 1,
      "orderDate" : ISODate("2017-03-09T16:30:16.415Z"),
      "orderStatus" : 0,
      "customerId" : 3
}
mongo>db.lineitems.findOne();
{
      "_id" : ObjectId("5a7935f97e9e82f6c6e77c2b"),
      "orderId" : 1,
      "prodId" : 158,
      "itemCount" : 48
}
mongo>db.products.findOne();
{
      "_id" : 1,
      "productName" : "Cup - 8oz Coffee Perforated",
      "price" : 56.92,
      "priceDate" : ISODate("2017-07-03T06:42:37Z"),
      "color" : "Turquoise",
      "Image" : "http://dummyimage.com/122x225.jpg/cc0000/ffffff"
}
```

In an embedded design, we would place absolutely all information relating to an order into a single document, as follows:

```
{
  "_id": 1,
  "first_name": "Rolando",
  "last_name": "Riggert",
  "email": "rriggert0@geocities.com",
  "gender": "Male",
  "Street": "6959 Melvin Way",
  "City": "Boston",
  "State": "MA",
  "ZIP": "02119",
  "SSN": "134-53-2882",
  "Phone": "978-952-5321",
  "Company": "Wikibox",
  "DOB": ISODate("1998-04-15T01:03:48Z"),
  "orders": [
    {
      "orderId": 492,
      "orderDate": ISODate("2017-08-20T11:51:04.934Z"),
      "orderStatus": 6,
      "lineItems": [
        {
          "prodId": 115,
          "productName": "Juice - Orange",
          "price": 4.93,
          "itemCount": 172,
          "test": true
        },
```

Each customer has their own document, and inside that document, there are an array of orders. Inside each order is an array of the products included in the order (line items) and all the information about a product contained within that line item.

In our example schema, there are 1000 customers, 1000 products, 51,116 orders, and 891,551 line items. The following indexes are defined:

```
OrderExample.embeddedOrders {"_id":1}
OrderExample.embeddedOrders {"email":1}
OrderExample.embeddedOrders {"orders.orderStatus":1}

OrderExample.customers {"_id":1}
OrderExample.customers {"email":1}

OrderExample.orders {"_id":1}
OrderExample.orders {"customerId":1}
OrderExample.orders {"orderStatus":1}

OrderExample.lineitems {"_id":1}
OrderExample.lineitems {"orderId":1}
OrderExample.lineitems {"prodId":1}
```

Let’s take a look at some typical operations that we might execute against these schemas and compare the performance for the two extremes.

Getting All the Data for a Customer

It’s a straightforward task to get all the data for a customer when all the information is embedded in a single document. We can get all the data from the embedded version with a query like this:

```
db.embeddedOrders.find({ email: 'bbroomedr@amazon.de' })
```

With an index on email, this query completes in less than a millisecond.

Life is much harder with the four-collection version. We need to use an aggregation or custom code to achieve the same result, and we need to be sure we have indexes on the `$lookup` join conditions (see Chapter 7). Here’s the aggregation:

```
db.customers.aggregate(
    [
      {
        $match: { email: 'bbroomedr@amazon.de' }
      },
      {
        $lookup: {
          from: 'orders',
          localField: '_id',
          foreignField: 'customerId',
          as: 'orders'
        }
      },
      {
        $lookup: {
          from: 'lineitems',
          localField: 'orders._id',
          foreignField: 'orderId',
          as: 'lineitems'
        }
      },
      {
        $lookup: {
          from: 'products',
          localField: 'lineitems.prodId',
          foreignField: '_id',
          as: 'products'
        }
      }
    ]
  )
```

Not surprisingly, the aggregation/join takes way longer than the embedded solution. Figure 4-2 illustrates the relative performance – the embedded model was able to deliver more than ten times more reads per second.

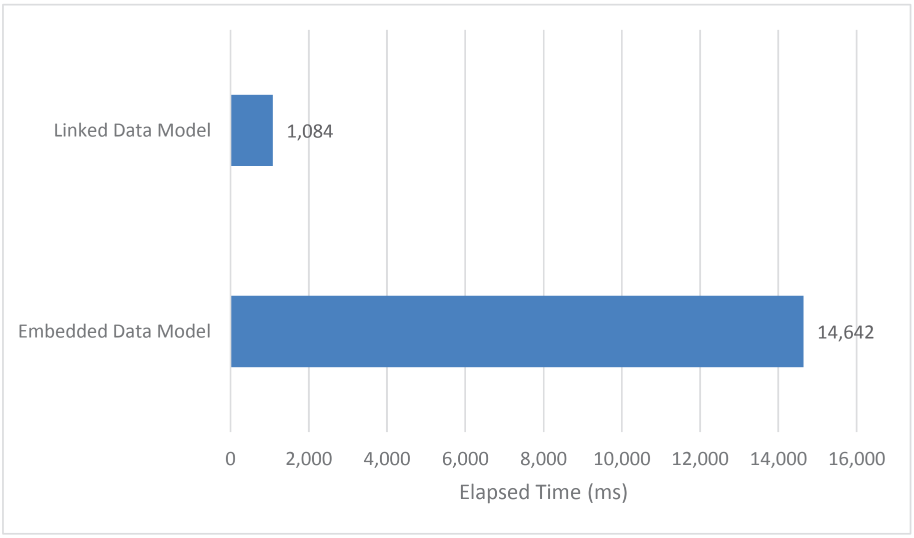


Figure 4-2. *Time taken to perform 500 customer lookups, including all order details*

Fetching All Open Orders

In a typical order processing scenario, we want to retrieve all the orders that are in an incomplete state. In our example, these orders are identified by `orderStatus=0`.

In the embedded case, we can get customers with open Orders like this:

```
db.embeddedOrders.find({"orders.orderStatus":0})
```

That does give us all customers with at least one open order, but if we only want to retrieve orders that are open, we are going to need to use the aggregation framework:

```
db.embeddedOrders.aggregate([
  { $match:{   "orders.orderStatus": 0 }},
  { $unwind:  "$orders" },
  { $match:{   "orders.orderStatus": 0 }},
  { $count: "count" }
] );
```

You might wonder why we have duplicate `$match` statements in our aggregation. The first `$match` gets us customers with open orders, while the second `$match` gets us the orders themselves. We don't need the first to get the right results, but it does improve performance (see Chapter 7).

It's far easier to get these orders in the linked data model:

```
db.orders.find({orderStatus:0}).count()
```

Not surprisingly, the simpler linked query gets the better performance. Figure 4-3 compares the performance of the two solutions.

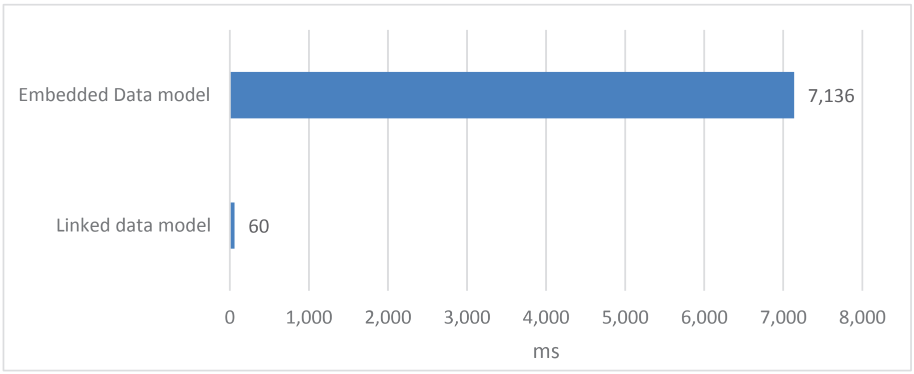


Figure 4-3. *Time taken to get a count of open orders*

Top Products

Most companies want to identify bestselling products. For the embedded model, we need to unwind the line items and aggregate by product name:

```
db.embeddedOrders.aggregate([
  { $unwind:  "$orders" },
  { $unwind:  "$orders.lineItems" },
  { $project: { "lineitems": "$orders.lineItems"   }},
  { $group:{  _id:{ "prodId":"$lineitems.prodId" ,
              " productName":"$lineitems.productName" },
              " itemCount-sum":{$sum:"$lineitems.itemCount"}} },
  { $sort:{  "lineitems_itemCount-sum":-1 }},
  { $limit:  10 },
]);
```

In the linked model, we also need to use aggregate, with $lookup joins between line items and products to get the product names:

```
db.lineitems.aggregate([
  { $group:{ _id:{ "prodId":"$prodId"  },
             "itemCount-sum":{$sum:"$itemCount"} }
  },
  { $sort:{  "itemCount-sum":-1 }},
  { $limit:  10 },
  { $lookup:
     { from:         "products",
       localField:   "_id.prodId",
       foreignField: "_id",
       as:           "product"
     }
  },
  { $project: {
          "ProductName": "$product.productName"   ,
          "itemCount-sum": 1   ,
          "_id": 1
         }
  },
]);
```

Despite having to perform a join, the linked data model performs best. We only have to join after we get the top ten products, while in the embedded design we have to scan all of the data in the collection. Figure 4-4 compares the two approaches. The embedded data model took about twice as long as the linked data model.

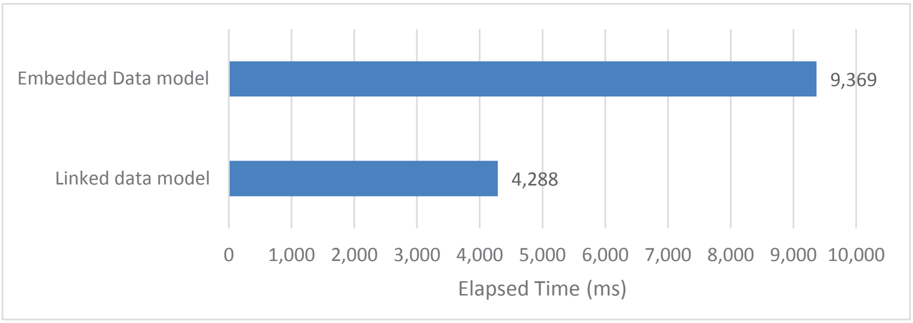


Figure 4-4. *Time taken to retrieve the top ten products*

Inserting New Orders

In this example workload, we looked at inserting a new order for an existing customer. In the embedded case, this is simply done by using a `$push` operation into the customer document:

```
db.embeddedOrders.updateOne(
            { _id: o.order.customerId },
            { $push: { orders: orderData } }
      );
```

In the linked data model, we have to insert into the `line items` collection and the `orders` collection:

```
var rc1 = db.orders.insertOne(orderData);
var rc2 = db.lineItems.insertMany(lineItemsArray);
```

You might think that the single update would easily outperform the multiple inserts required by the linked model. But actually, the update is a quite expensive operation – especially if there's not enough spare space in the collection to fit the new data. The linked inserts – though more numerous – are simpler operations because they don't require finding the matching document to update. Consequently, the linked model outperformed the embedded model for this example. Figure 4-5 compares the performance for 500 order inserts.

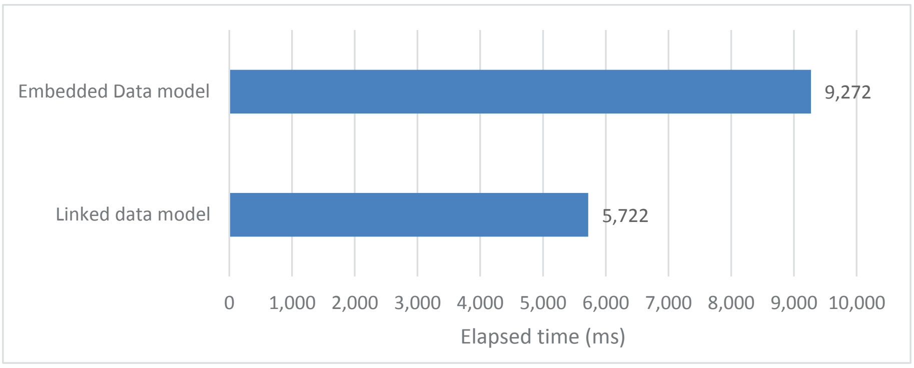


Figure 4-5. *Time to insert 500 orders*

Updating Products

What if we want to update the name of a product? In the embedded case, the product names are embedded into the line items themselves. We update the names of all the products in a single operation in MongoDB using the `arrayFilters` operator. Here, we update the name of product 193:

```
db.embeddedOrders.update(
      { 'orders.lineItems.prodId':193 },
      { $set: { 'orders.$[].lineItems.$[i].productName':
              'Potatoes - now with extra sugar' } },
      { arrayFilters: [{ 'i.prodId': { $eq: 193 } }], multi: true });
```

Of course, in the linked model, we can use a very simple update to the products collection:

```
db.products.update(
      { _id: 193 },
      { $set: { productName:  'Potatoes - now with extra sugar' } }
);
```

The embedded model requires us to touch many more documents than in the linked model. Consequently, ten product code price updates took hundreds of times longer in the embedded data model. Figure 4-6 illustrates the performance.

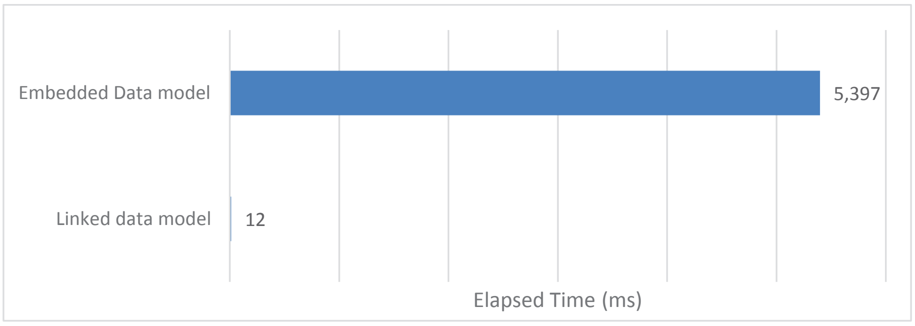


Figure 4-6. *Time to update ten product names*

Deleting a Customer

If we want to delete all data for a single customer in the four-collection model, we need to iterate through `line items`, `orders`, and `customers` collections. The code would look something like this:

```
db.orders.find({customerId:customerId},{_id:1}).forEach((order)=>{
      db.lineitems.deleteMany({orderId:order._id});
});
db.orders.deleteMany({customerId:1});
db.customers.deleteOne({_id:1});
```

Of course, in the embedded case, things are a lot easier:

```
db.embeddedOrders.deleteOne({_id:1});
```

The linked example performs very poorly – Figure 4-7 compares the performance for deleting 50 customers.

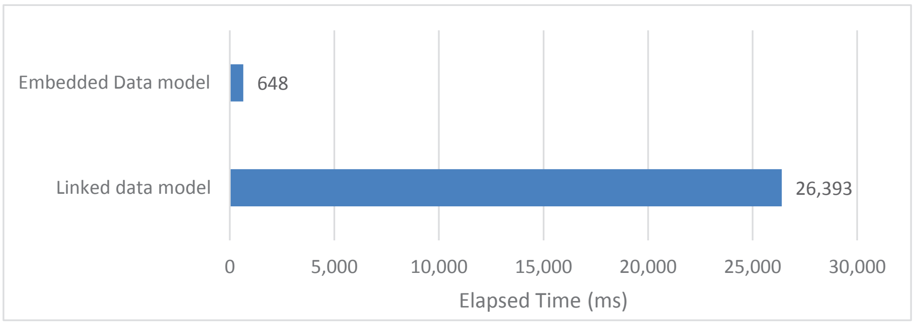


Figure 4-7. *Time to delete 50 customers*

Case Study Summary

We've looked at quite a few scenarios, and we wouldn't blame you if your head was spinning slightly. So let's aggregate all our performance data into one chart. Figure 4-8 combines the results from our six examples.

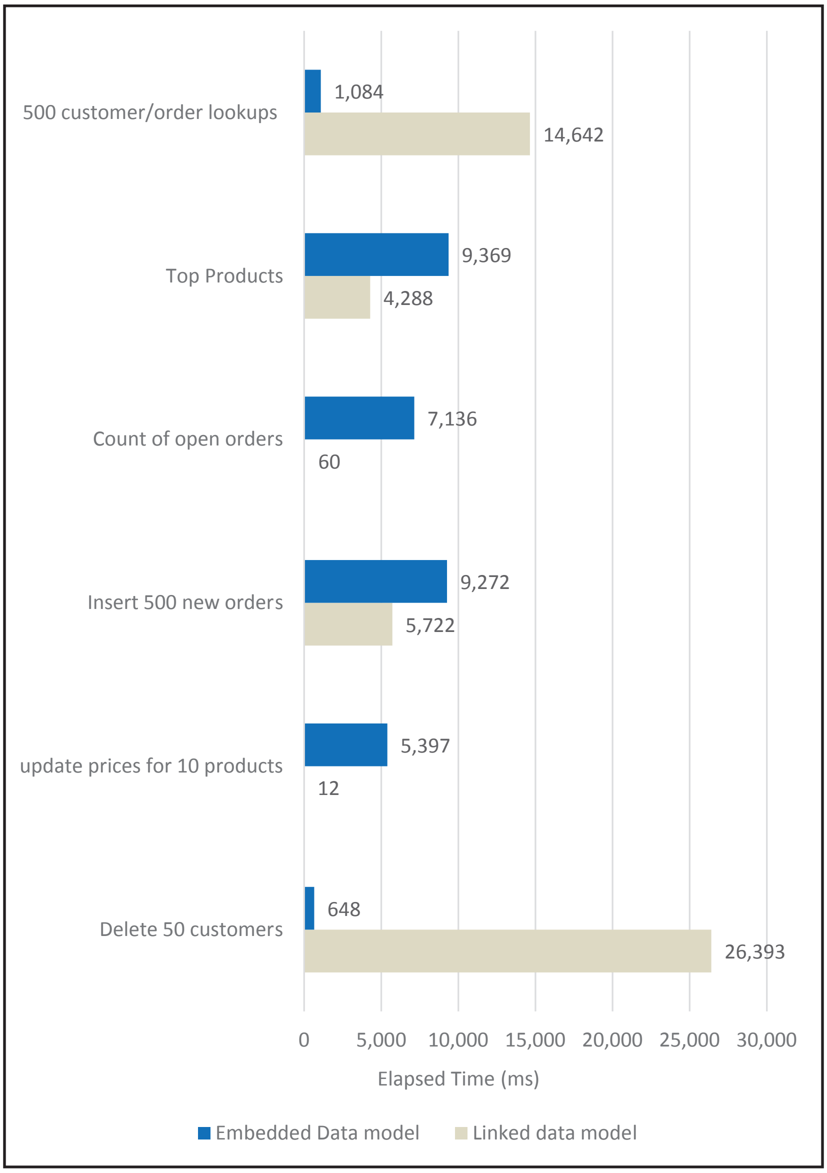


Figure 4-8. *Performance of linked vs. embedded models*

As you can see while the embedded model is pretty good at fetching all the data for a single customer or for deleting a customer, it's not superior to the linked alternative in other situations.

Tip The answer to the question "What is the best data model for my application" is – and always has been – "it depends."

The embedded model provides many advantages when reading all of the related data for an entity, but it is generally not the fastest model for updates and for aggregate queries. Which model works best for you will depend on which aspects of your application's performance are most critical. But remember, it's hard to change the data model once your application is deployed, so any time you spend getting your data model right early in the application design process will probably pay off.

Also, remember that very few applications use an "all or nothing" approach. The best outcomes are usually achieved when we mix linking and embedding approaches to maximize the critical operations for the application.

Advanced Patterns

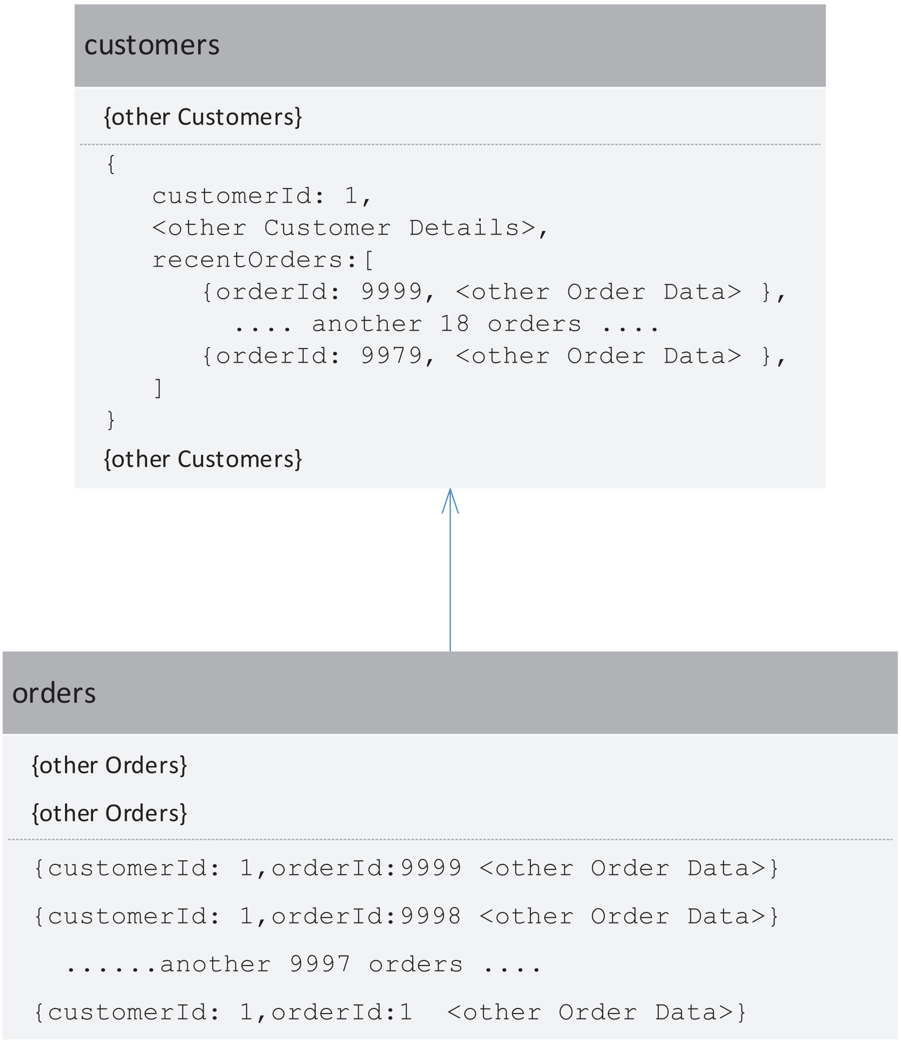
In the previous section, we looked at the two extremes of MongoDB data modelling: embedding everything vs. linking everything. In real life, you are likely to undertake a combination of both techniques to get the best balance between the trade-offs involved in each approach. Let's look at some of the modelling patterns that combine both approaches.

Subsetting

As we saw in the previous section, the embedded model has significant performance advantages when retrieving all data for an entity. However, there are two big risks that we need to be aware of:

- In a typical master-detail model – customers and their orders, for instance – the number of detail documents has no specific limit. But in MongoDB, a document must be no more than 16MB in size. So the embedded model can break if there are a large number of detail documents. For instance, our biggest customers might order so many products that we can't fit all the orders in a single 16MB document.

- Even if we are sure that the 16MB won't be exceeded, the effect on MongoDB memory might be undesirable. The number of documents that can fit into memory decreases as the average document size increases. Lots of large documents – potentially full of "old" data – might degrade the cache and reduce performance. We'll talk more about this in Chapter 11.
- One of the most common solutions to this conflict is a *hybrid* strategy, sometimes called *subsetting*.
- In the subsetting pattern, we embed a limited number of detail documents in the master document and store the remaining details in another collection. For instance, we might keep just the most recent 20 orders for each customer in the `customers` collection and the rest in an `orders` collection.
- Figure 4-9 illustrates the concept. Each customer has the most recent 20 orders embedded, with all orders available within the `orders` collection.



```
{other Customers}
{
   customerId: 1,
   <other Customer Details>,
   recentOrders:[
      {orderId: 9999, <other Order Data> },
        .... another 18 orders ....
      {orderId: 9979, <other Order Data> },
   ]
}
{other Customers}
```

```
{other Orders}
{other Orders}
{customerId: 1,orderId:9999 <other Order Data>}
{customerId: 1,orderId:9998 <other Order Data>}
  ......another 9997 orders ....
{customerId: 1,orderId:1  <other Order Data>}
```

Figure 4-9. *A hybrid "bucket" data model*

If we imagine that our application displays the most recent orders for each customer on a customer lookup page, then we can see the benefits of this model. Not only have we avoided hitting the 16M document size limit, but we can now populate this customer lookup page from a single document.

However, the solution does come at a cost. In particular, we now have to shuffle orders in the embedded orders array every time we add or modify an order. Each update would need to perform additional manipulation of the embedded orders. The following code implements the shuffle of `customers` data in the hybrid design:

```
let orders=db.hybridCustomers.
             findOne({'_id':customerId}).orders;

orders.unshift(newOrder); // add new order
if (orders.length>20)
  orders.pop();            // Remove the order
db.hybridCustomers.update({'_id':customerId},
        {$set:{orders:orders}});
```

The resulting overhead can be significant. Figure 4-10 shows the impact of the hybrid model when fetching customers and most recent orders and when updating customers with new orders. Read performance was significantly improved, but the update rate was almost halved.

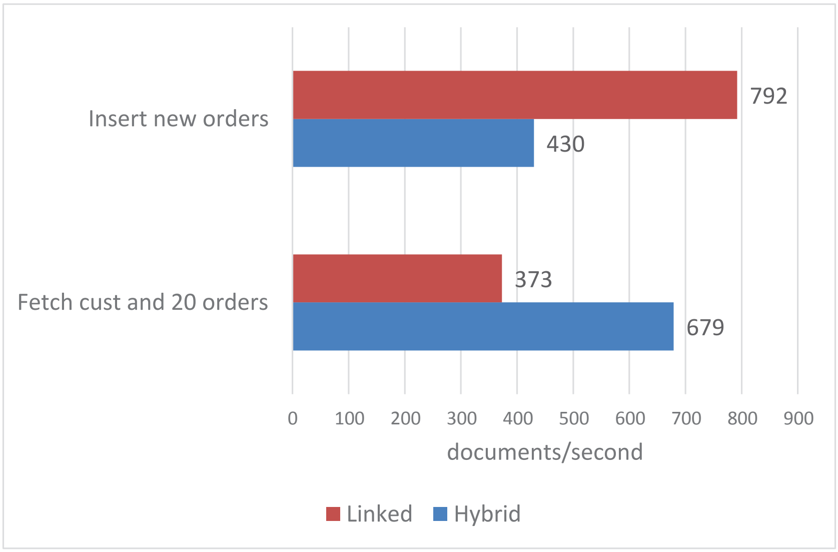


***Figure 4-10.** The hybrid model can improve read performance, but slow down updates*

Vertical Partitioning

It generally makes sense to put everything relating to an entity in a single document. As we've seen previously, we can embed the multiple details relating to an entity in a JSON array, avoiding what would have required a join operation in a SQL database.

However, sometimes we can get benefits from splitting the details for an entity across multiple collections so that we can reduce the amount of data fetched in each operation. This approach is similar to the hybrid data model in that it reduces the size of the core document, but it is applied to top-level attributes, not just to arrays of details.

For instance, imagine that in each customer record we include a high-resolution photograph of the customer. These infrequently accessed images increase the overall size of the collection, degrading the time taken to perform collection scans (see Chapter 6). They also reduce the number of documents that can be held in memory which might increase the amount of IO required (see Chapter 11).

In this scenario, we can get a performance advantage if we store the binary photos in a separate collection. Figure 4-11 illustrates the arrangement.

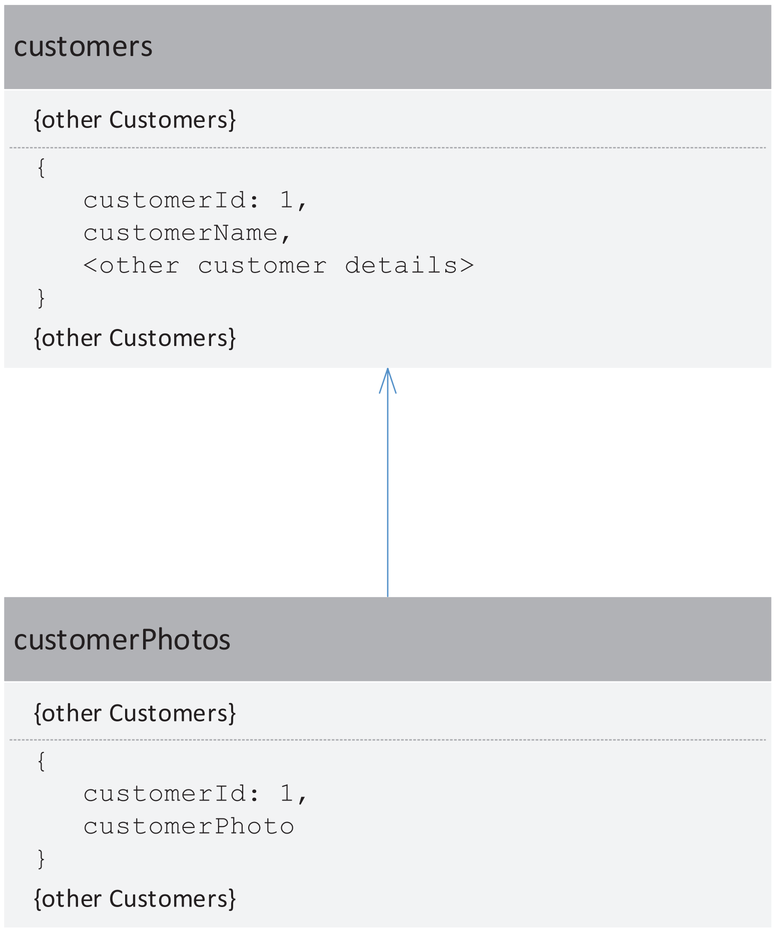


Figure 4-11. *Vertical partitioning*

The Attribute Pattern

If we have documents that include a large number of attributes of the same data type, and we know that we are going to be performing lookups using a many of these attributes, then we can reduce the number of indexes we need by using the attribute pattern.

Consider the following weather data:

```
{
        "timeStamp" : ISODate("2020-05-30T07:21:08.804Z"),
        "Akron" : 35,
        "Albany" : 22,
        "Albuquerque" : 22,
        "Allentown" : 31,
        "Alpharetta" : 24,
        <data for another 300 cities>
}
```

If we know that we will be supporting queries that search for specific values for a city (find all measurements over 100 degrees in Akron, for instance), then we have a problem. We can't possibly create enough indexes to support all the queries. A better organization would be to define `name:value` pair for each city.

Here's how the preceding data would look like in the attribute pattern:

```
{
        "timeStamp" : ISODate("2020-05-30T07:21:08.804Z"),
        "measurements" : [
                {
                        "city" : "Akron",
                        "temperature" : 35
                },
                {
                        "city" : "Albany",
                        "temperature" : 22
                },
                {
                        "city" : "Albuquerque",
                "temperature" : 22
        },
        {
                "city" : "Allentown",
                "temperature" : 31
        },
         <data for another 300 cities>
}
```

We now have the option to define a single index on `measurements.city`, rather than attempting the impossible task of creating hundreds of indexes which would have been needed in the first design.

In some cases, you can use wildcard indexes rather than the attribute pattern – see Chapter 5. Nevertheless, the attribute pattern provides a flexible way to provide fast access to arbitrary data items.

Summary

Although MongoDB supports very flexible schema modelling, your data model design remains absolutely critical to application performance. The data model determines the amount of logical work that MongoDB needs to perform to satisfy database requests and can be very difficult to change once deployed to production.

The two "meta-patterns" in MongoDB modelling are *embedding* and *linking*. Embedding involves including all information about a logical entity in a single document. Linking involves storing related data in separate collections in a manner reminiscent of relational databases.

Embedding improves read performance by avoiding joins but can create challenges involving data consistency, update performance, and the 16MB document limit. Most applications mix embedding and linking judiciously to achieve a "best of both worlds" solution.