NoSQL

...

into the great beyond

Where do we go from here?

Our previous module showed one way to implement a data model using a specific storage mechanism: a relational database, where the model is decomposed into a relational schema: tables with columns, using primary keys and foreign keys to enforce constraints on the data.

But that is our *second* data storage mechanism, after JSON flat files...

So let's compare JSON to RDBMs as a data storage layer...

	JSON flat files	RDBMs
Store a sophisticated data model		
Enforce complicated schemas		
Efficiently find data		
Efficiently mutate data		
Multi-user access & security		
Robustness		

Criticisms of the relational model

- Joins are slow. The relational model uses joins to find related objects, which is *pretty important* in data modeling.
- Designing a table requires a different skill set from designing OO classes.
- Schemas (table designs) are inflexible; difficult to change the columns of a table if it is already full of data.
- Horizontal scaling is difficult.

Which of these are especially problematic in the tech industry?

NoSQL

"NoSQL" – which really means *not only SQL* – refers to any database that does not use the relational model. These alternative models *generally*:

- Don't use joins; leave it up to the application to request and join related objects.
- Don't require strict schemas.
  - There are often no "types" at all, just objects with fields.
- Don't promise consistency or reliability.
  - BASE: Basically Available, Soft-state, Eventually consistent.

By reducing some of the strictness and correctness of the relational model, we might end up with a system that is more flexible, scalable, and easier to work with.

NoSQL models

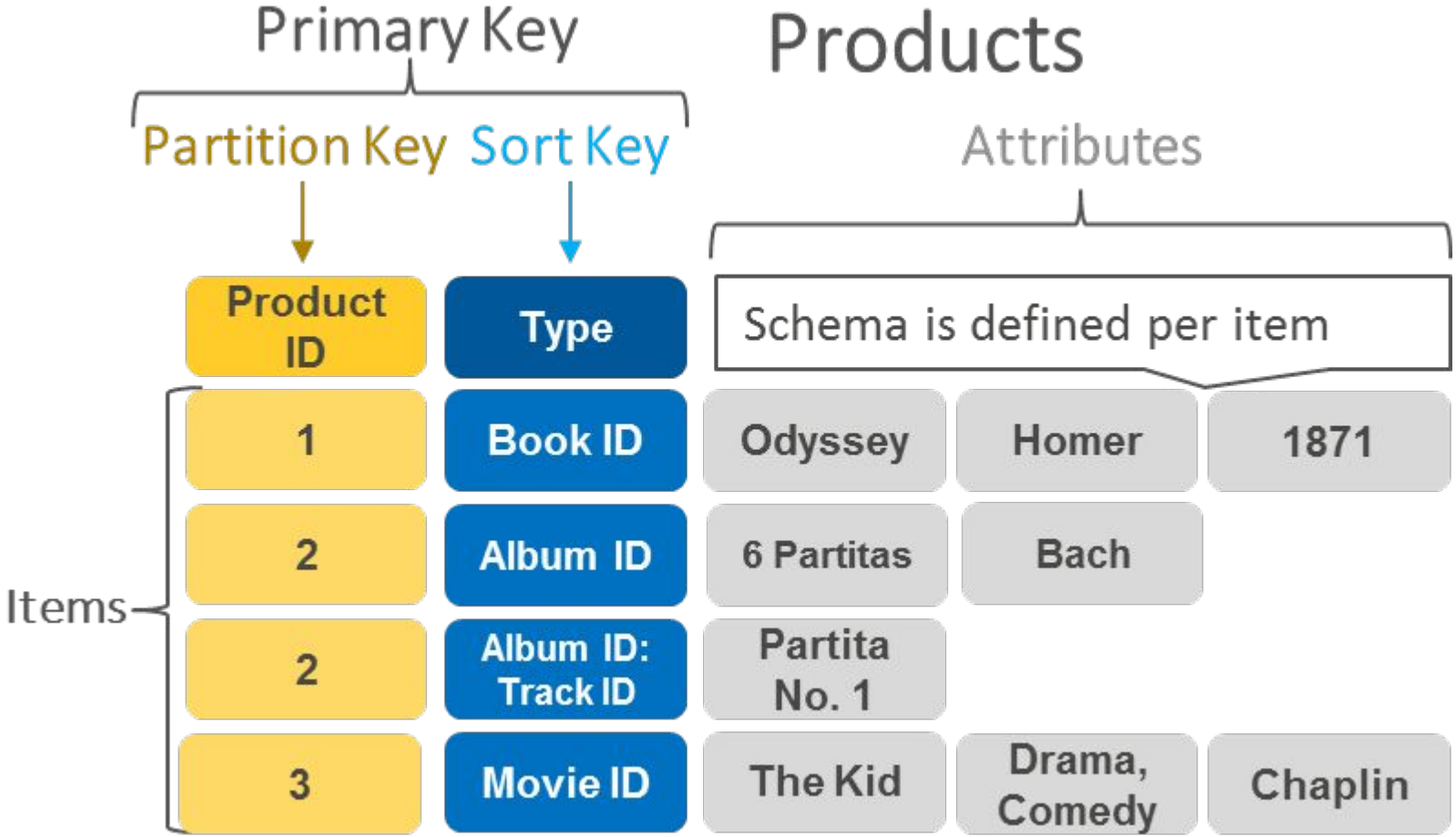
There is no single definition of “NoSQL”; any non-relational model technically fits in this umbrella. These are some of the more popular NoSQL models:

- Key-value store
  - Amazon DynamoDB, Redis
- Document store
  - MongoDB, Google Firebase, Microsoft Azure Cosmos DB
- Graph
  - Neo4J

Key-value stores

Suppose we have an *enormous collection of objects* that are all **isolated**: they don't have associations with other objects. We can uniquely identify each object with a primary key, and need to retrieve them frequently by that key, but (generally) not by any other condition.

There are too many objects and too many requests for them for a single server to handle. We don't need the complexity of SQL, so we drop the relational model completely. A **key-value store** is a database built for this use case: *tons* of data moving both directions, identified only by primary keys, scaled horizontally using partitioning.


Primary Key
Products
Partition Key
Sort Key
Attributes
Product ID
Type
Schema is defined per item
Items
1
Book ID
Odyssey
Homer
1871
2
Album ID
6 Partitas
Bach
2
Album ID: Track ID
Partita No. 1
3
Movie ID
The Kid
Drama, Comedy
Chaplin

occasion	giver	gifts	recipient_react...	thank_you_...
ada birthday	Jaclyn	[{ "S" : "pete the cat book" }, { "S" : "easel" }]		
ada birthday	Neal	[{ "S" : "Yoshi's Crafted World" }, { "S" : "Hot Wheels" }]	overjoyed	
madison birthday	Ada	[{ "S" : "Classical music Tonie" }]		
madison birthday	Neal			
madison birthday	Sandee	[{ "S" : "onesie" }, { "S" : "bottle" }]		
madison birthday	Tom	[{ "S" : "stuffy manatee" }]		false

Key-value summary

1. Objects can only be efficiently retrieved by their primary key.

2. There is no schema beyond the primary key. Your application must know what type of data you are expecting and do its own validation.

3. There are no foreign keys. Joins can be done application-side, but that is slow.

4. Since we only ever retrieve one item at a time, and there are no joins, it doesn't matter which physical server/partition an object "lives" on. We can scale almost infinitely using partitioning.

Key-value store use cases

1. In-memory caching
    a. Redis and memcached

2. TONS of simple, isolated objects
    a. There is no joining of items in a KV database, so don't use one if you have complex objects that reference each other.
    b. Usually optimized for single-item retrieval, so don't use one if you frequently need to find many objects that satisfy some condition.
    c. Primary keys are used to identify which partition the object is on, in constant time. This database scales extremely well with partitioning.

Live Demo: Python with AWS DynamoDB

```
Enter an event name: ada birthday
{'Items': [{'giver': 'Jaclyn', 'gifts': ['pete the cat book', 'easel'], 'occasion': 'ada birthday'}, {'giver':
'Neal', 'gifts': ["Yoshi's Crafted World", 'Hot Wheels'], 'occasion': 'ada birthday', 'recipient_reaction': 'ov
erjoyed'}], 'Count': 2, 'ScannedCount': 2, 'ResponseMetadata': {'RequestId': 'BGJINJUDP2MDNPKTMLPM2FHD07VV4KQNS
O5AEMVJF66Q9ASUAAJG', 'HTTPStatusCode': 200, 'HTTPHeaders': {'server': 'Server', 'date': 'Mon, 14 Nov 2022 21:1
5:26 GMT', 'content-type': 'application/x-amz-json-1.0', 'content-length': '310', 'connection': 'keep-alive', '
x-amzn-requestid': 'BGJINJUDP2MDNPKTMLPM2FHD07VV4KQNSO5AEMVJF66Q9ASUAAJG', 'x-amz-crc32': '1632195203'}, 'Retry
Attempts': 0}}
```


```
{
    "Items": [
        {
            "giver": "Jaclyn",
            "gifts": ["pete the cat book", "easel"],
            "occasion": "ada birthday"
        },
        {
            "giver": "Neal",
            "gifts": ["Yoshi's Crafted World", "Hot Wheels"],
            "occasion": "ada birthday",
            "recipient_reaction": "overjoyed"
        }
    ],
    "Count": 2,
    "ScannedCount": 2,
    ...
```

Document stores

Document stores

We define a named *collection*, like “students” or “courses”. We then insert *documents* into the collection. A document is a JSON object, with fields that have values. A collection then contains many objects.

Documents can be as simple or complex as JSON allows, including the ability to “embed” children directly inside a parent object.

```
{
    "_id": "S10_1678",
    "productname": "1969 Harley Davidson",
    "productline": "Motorcycles",
    ...
    "quantityinstock": 7933,
    "buyprice": 48.81,
    "msrp": 95.7
}
```

```
{
    "_id": 103,
    "customername": "Atelier graphique",
    ...
    "payments": [
      {
        "checknumber": "HQ336336",
        "amount": 6066.78,
        ...
      },
      ...
    ],
```

Intro to MongoDB

MongoDB is probably the most well-known NoSQL document store database. A Mongo database contains one or more **collections** of **documents**, which are stored internally as JSON objects. A special query language is used to find documents based on their contents.

Every document in Mongo has a field called **"_id"**, which serves as the document's unique identifier (primary key). An _id can be any data type, including complex objects (for multi-value keys). If you don't provide one, Mongo will automatically assign a random 12-byte integer called an ObjectId. Documents can be retrieved quickly based on their _id field, and *possibly* also based on other fields (later)...

Live Demo: Brief walkthrough of Compass

MongoDB data types

MongoDB technically uses a variant of JSON called “BSON”. It supports the data types from JSON: the primitives int, float, string, bool; also “array” and “object”. An object-like syntax is used for other “built in” types, like timestamps and “ObjectID”.

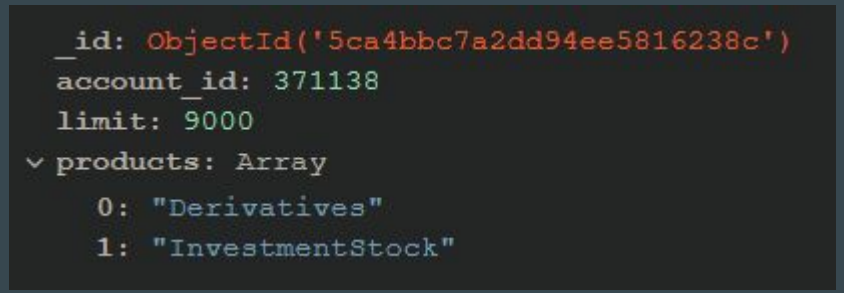
```
{
  "checknumber": "HQ336336",
  "paymentdate": {
    "$date": "2014-10-19"
  },
  "amount": 6066.78
},
```

```
{
  "_id": {
    "$oid": "5ca4bbcea2dd94ee58162a68"
  },
  "username": "fmiller",
  "name": "Elizabeth Ray",
  "address": "9286 Bethany Glens\nVasqueztown, CO 22939",
  "active": true
}
```

Training database: accounts

We will use a training database representing customers and accounts at a brokerage firm. The first collection (table) in this database is **accounts**; here is one document:

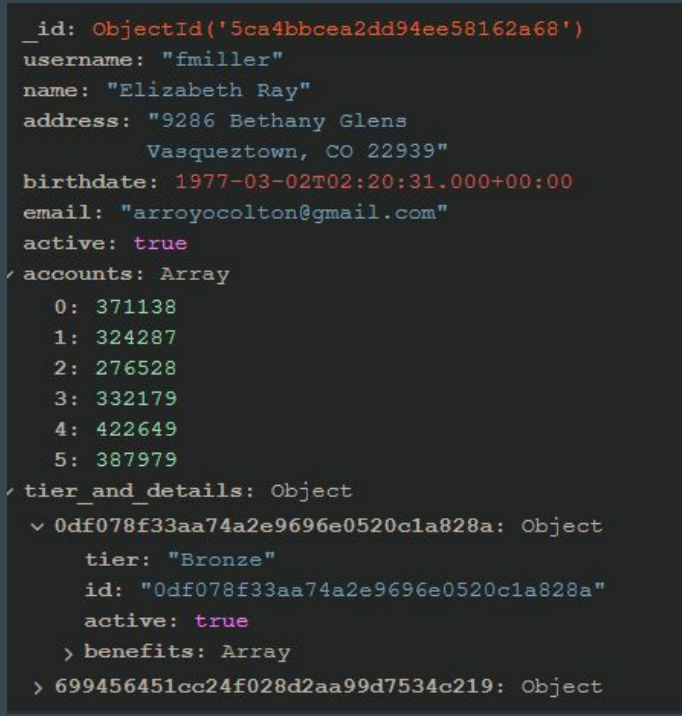
```
{
  "_id": {
    "$oid": "5ca4bbc7a2dd94ee5816238c"
  },
  "account_id": 371138,
  "limit": 9000,
  "products": [
    "Derivatives",
    "InvestmentStock"
  ]
}
```



Training database: customers

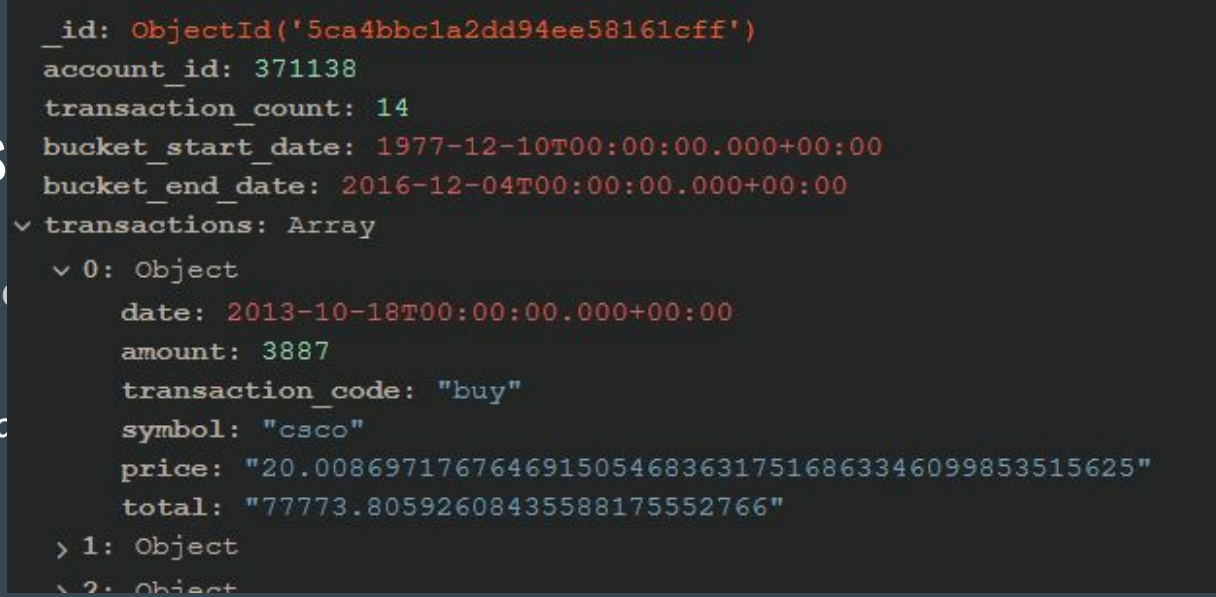
The **customers** collection identifies each customer and the accounts they own.

```
{
  "_id": {
    "$oid": "5ca4bbcea2dd94ee58162a68"
  },
  "username": "fmiller",
  "name": "Elizabeth Ray",
  "accounts": [371138, 324287, … for brevity … ],
  "tier_and_details": {
    "0df078f33aa74a2e9696e0520c1a828a": {
      "tier": "Bronze",
      "id": "0df078f33aa74a2e9696e0520c1a828a",
      "active": true,
      "benefits": ["sports tickets"]
    }
  }
}
```



Training database: trans

The **transactions** collection hol




```
{
  "_id": { "$oid": "5ca4bbc1a2c
  "account_id": 371138,
  "transaction_count": 14,
  "transactions": [
    {
      "date": { "$date": { "$numberLong": "1382054400000" } },
      "amount": 3887,
      "transaction_code": "buy",
      "symbol": "csco",
      "price": "20.008697176764691505468363175168633460998535156 25",
      "total": "77773.8059260843558817555 2766"
    }, …
  ]
}
```

Schemas for document databases

Document databases usually don’t require schemas for their collections, bu they do recommend them. MongoDB uses a variant of the “JSON Schema” specification put constraints on the documents that are placed in a collection. The syntax is fairly easy...

```
{
    "$jsonSchema": {
        "title": "products",
        "description": "A product sold by the company",
        "bsonType": "object",
        "required" : ["_id", "productname", "buyprice", ...],
        "properties": {
            "_id" : {
                "bsonType" : "string"
            },
            "productname": {
                "bsonType": "string"
            },
            "buyprice": {
                "bsonType": "float",
                "minimum": 0.0
            },
            ...
        }
    }
}
```

Designing document schemas

Document schema guidelines

A **relational** database schema typically aims to minimize redundancy and maximize data integrity through normalization. This leads to lots of small, narrowly-focused tables with foreign keys to other tables. But this means lots of joins, which are slow.

Designing collections in a document database is more of an art form, where we balance several competing concerns. In this class, we will focus on two major decisions:

- Should children be **embedded** or **referenced**?
- When is redundancy actually OK?

Embedding vs. referencing

Compare these three document designs:

```
// orders collection
{
    "_id" : 1,
    "orderDate" : "2022-11-16T17:00:00",
    "deliveryAddress" : "1600 Bellflower Blvd,
                         Long Beach, CA 90840"
}

// pizzas collection
{
    "_id" : 1,
    "orderNumber" : 1,
    "size" : "small"
}

{
    "_id" : 2,
    "orderNumber" : 1,
    "size" : "small"
}

      size  :  small
 }
```

Essentially a relational database schema.

```
// orders collection
{
    "_id" : 1,
    "orderDate" : "2022-11-16T17:00:00",
    "deliveryAddress" : "1600 Bellflower Blvd,
                         Long Beach, CA 90840",
    "pizzas" : [1, 2]
}

// pizzas collection
{
    "_id" : 1,
    "size" : "small"
}

{
    "_id" : 2,
    "size" : "small"
}

   }
```

Order references its children.

```
// orders collection
{
    "_id" : 1,
    "orderDate" : "2022-11-16T17:00:00",
    "deliveryAddress" : "1600 Bellflower Blvd,
                         Long Beach, CA 90840",
    "pizzas" : [
        {
            "_id" : 1,
            "size" : "small"
        },
        {
            "_id" : 2,
            "size" : "small"
        }
    ]
}
```

Order embeds its children.

Embedding vs. referencing, pt 2

We have three choices for implementing a parent-child relationship:

1. Parent embeds child.
    a. Don't have to use a join to unite a parent with its children!
    b. Can add an index on the fields of the embedded children, so you can still search them quickly.
    c. Document databases usually put a max size on documents; in Mongo, it is 16MB.
2. Parent references ID of child.
    a. Can fit many more child IDs than child objects into one document.
    b. Reading the parent document at least tells you the # of children and their IDs.
    c. Must use a join to match a parent with its children.
    d. With LOTS of children, can still run into document max size limit.
3. Child references ID of parent.
    a. Can have an infinite number of children now, since the parent doesn't store anything about them.
    b. Cannot use a parent document to know anything about its children.
    c. Must use a join to match a parent with its children.

Implementing one-to-many associations

With the lessons from the previous slide, Mongo developers break one-to-many associations into three sub-categories, each with their own implementation:

1. One-to-few: a parent has a few children. Embed them in the parent.

2. One-to-many: a parent has too many children to embed. Parent stores a list of child IDs. Children can optionally store their parent ID.

3. One-to-**squillions**: too many children to even store a list of their IDs. Parent loses all knowledge of children; instead, the children store the ID of their parent.

```
// orders collection
{
    "_id" : 1,
    "orderDate" : "2022-11-16T17:00:00",
    "deliveryAddress" : "1600 Bellflower Blvd,
                         Long Beach, CA 90840"
}

// pizzas collection
{
    "_id" : 1,
    "orderNumber" : 1,
    "size" : "small"
}

{
    "_id" : 2,
    "orderNumber" : 1,
    "size" : "small"
}
```

One-to-squillions.

```
// orders collection
{
    "_id" : 1,
    "orderDate" : "2022-11-16T17:00:00",
    "deliveryAddress" : "1600 Bellflower Blvd,
                         Long Beach, CA 90840",
    "pizzas" : [1, 2]
}

// pizzas collection
{
    "_id" : 1,
    "size" : "small"
}

{
    "_id" : 2,
    "size" : "small"
}
```

One-to-many.

```
// orders collection
{
    "_id" : 1,
    "orderDate" : "2022-11-16T17:00:00",
    "deliveryAddress" : "1600 Bellflower Blvd,
                         Long Beach, CA 90840",
    "pizzas" : [
        {
            "_id" : 1,
            "size" : "small"
        },
        {
            "_id" : 2,
            "size" : "small"
        }
    ]
}
```

One-to-few.

Many-to-many associations

Similar decisions are used to implement many-to-many associations, with the additional complexity of dealing with the association class / junction table, if any.

If there is **no association class**, then each end of the many-to-many can reference the objects it is related to, as if it's just a one-to-many association.

```
// films collection
{
    "_id" : 15,
    "title" : "Black Panther: Wakanda Forever",
    "genres" : [1, 4, 13]
}

// genres collection
{
    "_id" : 1,
    "name" : "Action",
    "films" : [15, 64, 99, 102, 105]
}
```

```
> db.films.find({"_id" : 15}, {"genres" : 1})
< { _id: 15, genres: [ 1, 4, 13 ] }
```

```
> db.genres.find({"name" : "Action"}, {"films" : 1})
< { _id: 1, films: [ 15, 64, 99, 102, 105 ] }
```

Many-to-many associations, pt. 2

With an **association class**, a new collection is needed to store the association/junction objects. Unless we have a “many to squillions” design, the two parent objects can reference the junction objects using a list.

```
// tickets collection
{
    "_id" : 323,
    "showingId" : 999,
    "seatId" : 105,
    "price" : 2499
}
// showings collection
{
    "_id" : 999,
    "filmId" : 15,
    "roomId" : 42,
    "showTime" : { "$date" : "2022-11-28T17:00:00" },
    "tickets" : [323, 356, 387, 388, 389]
}
```

Tickets is the junction collection, referencing the IDs of showings and seats.

A showing has many tickets, so ticket IDs are referenced in a list.

A seat has squillions of tickets, so the parent object (seat) does not reference its children (tickets) at all.

Mongo Queries

MongoDB query syntax

No matter the context, working with data always requires a language that can:

1. Choose which collection of data to examine
2. Identify which data elements to keep (filtering)
3. Group the data
4. Select which fields and aggregates to output or transform (projection)
5. Sort the data
6. Page the data (limit & offset)

MongoDB does not use SQL; instead it uses a more traditional set of function calls to perform these operations.

Basic selection

To find one (the first) document in a collection, we use findOne():

```
> db.products.findOne()
<   {
      _id: 'S10_1678',
      productname: '1969 Harley Davidson Ultimate Chopper',
      productline: 'Motorcycles',
      productscale: '1:10',
      productvendor: 'Min Lin Diecast',
      productdescription: 'This replica features working kickstand, front suspension,
      quantityinstock: 7933,
      buyprice: 48.81,
      msrp: 95.7
    }
```

There are many other functions like this for finding, sorting, inserting, updating, and deleting data.

But we are going to learn an alternate form of querying...

Basic selection, pt 2

To select all documents, use find():

```
> db.accounts.find()
< { _id: ObjectId("5ca4bbc7a2dd94ee5816238c"),
    account_id: 371138,
    limit: 9000,
    products: [ 'Derivatives', 'InvestmentStock' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee5816238d"),
    account_id: 557378,
    limit: 10000,
    products: [ 'InvestmentStock', 'Commodity', 'Brokerage', 'CurrencyService' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee5816238e"),
    account_id: 198100,
    limit: 10000,
    products: [ 'Derivatives', 'CurrencyService', 'InvestmentStock' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee5816238f"),
```

Basic filtering

To filter the document(s) returned, we pass an argument to find()/findOne() identifying some condition to filter the results. This can get **super complicated**.

For equality:

```
> db.accounts.findOne({"account_id" : 137994})
< { _id: ObjectId("5ca4bbc7a2dd94ee5816239f"),
    account_id: 137994,
    limit: 10000,
    products: [ 'CurrencyService', 'InvestmentStock' ] }
```

```
SELECT *
FROM accounts
WHERE account_id = 137994
FETCH FIRST 1 ROWS ONLY
```

Inequality:

```
> db.accounts.findOne({"account_id" : {"$gt" : 400_000}})
< { _id: ObjectId("5ca4bbc7a2dd94ee5816238d"),
    account_id: 557378,
    limit: 10000,
    products: [ 'InvestmentStock', 'Commodity', 'Brokerage', 'CurrencyService' ] }
```

```
SELECT *
FROM accounts
WHERE account_id >= 400000
FETCH FIRST 1 ROWS ONLY
```

Basic filtering, pt 2

To filter documents that have a list containing some value, treat it like equality:

```
> db.accounts.find({"products" : "Commodity"})
< { _id: ObjectId("5ca4bbc7a2dd94ee5816238d"),
    account_id: 557378,
    limit: 10000,
    products: [ 'InvestmentStock', 'Commodity', 'Brokerage', 'CurrencyService' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee58162390"),
    account_id: 278603,
    limit: 10000,
    products: [ 'Commodity', 'InvestmentStock' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee58162391"),
    account_id: 383777,
    limit: 10000,
    products:
     [ 'CurrencyService',
       'Derivatives',
       'InvestmentFund',
       'Commodity',
```

In relational databases, we couldn't have a column store an array of values. If we could, our query would be:

SELECT *
FROM accounts
WHERE 'Commodity' IN products

In reality, we'd need products to be their own table, and a many-to-many junction to join them to accounts.

SELECT *
FROM accounts a
INNER JOIN account_products ap ON a.account_id = ap.account_id
INNER JOIN products p ON p.product_id = ap.product_id
WHERE p.productname = 'Commodity'

Basic projection

To choose the fields for output, we pass a SECOND argument to findOne()/find():

```
> db.accounts.find({"products" : "Commodity"}, {"products" : 1})
< { _id: ObjectId("5ca4bbc7a2dd94ee5816238d"),
    products: [ 'InvestmentStock', 'Commodity', 'Brokerage', 'CurrencyService' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee58162390"),
    products: [ 'Commodity', 'InvestmentStock' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee58162391"),
    products:
     [ 'CurrencyService',
       'Derivatives',
       'InvestmentFund',
       'Commodity',
       'InvestmentStock' ] }
  { _id: ObjectId("5ca4bbc7a2dd94ee58162399"),
    products: [ 'CurrencyService', 'Brokerage', 'InvestmentStock', 'Commodity' ] }
```

{“products” : 1} says to only keep the products field.

{“products” : 1, “account_id” : 1} would keep account_id also.

_id is always kept, unless you add “_id” : 0 to the projection map.

SQL (with fake “array” column):
SELECT products
FROM accounts
WHERE ‘Commodity’ IN products

Basic sorting

To sort, a .sort() function call is added after the find()/findOne(), indicating the field(s) to sort by.

```
> db.customers.find({}, {"username" : 1}).sort({"username" : 1})
< { _id: ObjectId("5ca4bbcea2dd94ee58162a95"),
    username: 'abrown' }
  { _id: ObjectId("5ca4bbcea2dd94ee58162b70"),
    username: 'alexandra72' }
  { _id: ObjectId("5ca4bbcea2dd94ee58162b49"),
    username: 'alexsanders' }
  { _id: ObjectId("5ca4bbcea2dd94ee58162ad4"),
    username: 'allenhubbard' }
  { _id: ObjectId("5ca4bbcea2dd94ee58162ade"),
    username: 'allenjennifer' }
  { _id: ObjectId("5ca4bbcea2dd94ee58162bce"),
    username: 'alvarezdavid' }
  { _id: ObjectId("5ca4bbcea2dd94ee58162bfb"),
    username: 'amanda41' }
```

No filter condition. Project onto username only. Sort by username in ascending order. (use -1 for desc.)

In SQL:
SELECT username
FROM customers
ORDER BY username

Pipelines

A **pipeline** is a general software term for a sequence of operations, where the output of one **stage** in the sequence becomes the input to the next stage.

A SQL query is a pipeline. Identify the input and output of each of these clauses:

1. FROM
2. JOIN
3. WHERE
4. GROUP BY
5. HAVING
6. SELECT
7. ORDER BY
8. LIMIT

The SQL SELECT pipeline always executes in a fixed order. But what if we want to break that order... say, by grouping by a column that we create in SELECT? Subqueries were invented to solve this problem, but they can get very complicated, very fast.

What might be better is if we could choose the order of the pipeline stages...

MongoDB pipeline queries

MongoDB has a very flexible querying function called **aggregate()**. Given a list of **pipeline stages**, aggregate() executes each stage in order, sending the output of a stage to be the input of the following stage.

```
db.products.aggregate([
    // stage 1,
    // stage 2,
    // stage 3,
    // etc.
])
```

"db.products" is equivalent to FROM (without any joins). The [] list inside aggregate() defines the stages to execute in order.

Each stage is defined as an object with a key identifying the stage function, and a value of the parameters to give that function.

```
// template of a stage object
{
    "$functionName": {
        // function parameters go here

    }
}
```

Pipeline stages: $project

The $project function specifies a projection: which fields to keep from each input.

```
db.products.aggregate([
  {
    "$project": {
      "productname": 1
    }
  }
])
```

```
{
  _id: 'S10_1678',
  productname: '1969 Harley Davidson Ultimate Chopper'
}
{
  _id: 'S10_1949',
  productname: '1952 Alpine Renault 1300'
}
{
  _id: 'S10_2016',
  productname: '1996 Moto Guzzi 1100i'
}
```


{
_id: 'S10_1678',
productname: '1969 Harley Davidson Ultimate Chopper',
productline: 'Motorcycles',
productscale: '1:10',
productvendor: 'Min Lin Diecast',
productdescription: 'This replica features working ki
quantityinstock: 7933,
buyprice: 48.81,
msrp: 95.7
}
"$project": {
"productname": 1
}
{
_id: 'S10_1678',
productname: '1969 Harley Davidson Ultimate Chopper'
}

```
db.products.aggregate([
  {
    "$project": {
      "productname": 1,
      "productline": 1,
      "_id": 0
    }
  }
])
```

```
{
  productname: '1969 Harley Davidson Ultimate Chopper',
  productline: 'Motorcycles'
}
{
  productname: '1952 Alpine Renault 1300',
  productline: 'Classic Cars'
}
{
  productname: '1996 Moto Guzzi 1100i',
  productline: 'Motorcycles'
}
{
  productname: '2003 Harley-Davidson Eagle Drag Bike',
  productline: 'Motorcycles'
}
```

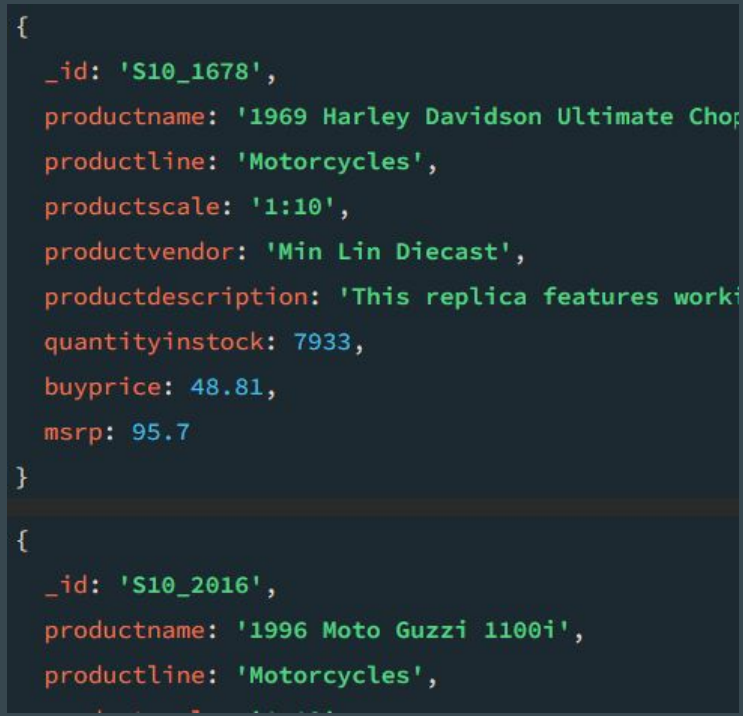
Challenge #1: select only the customername, city, and country of all customers.

Pipeline stages: $match

The $match function produces a new set of output by filtering the input rows.

```
db.products.aggregate([
  {
    "$match": {
      "productline": "Motorcycles"
    }
  }
])
```

Only products with a productline **equal to** “Motorcycles” will be kept.



Challenge #2: select only customers that are in “USA”.

More complicated matches

Matching equality is easy. Other comparisons are trickier.

```
db.products.aggregate([
  {
    "$match": {
      "productline": { "$ne": "Motorcycles" }
    }
  }
])
```

{ “$ne” : ... } is an **operator function**. Imagine each object’s productline field being inserted left of the “not equals” function, i.e., “productline != ‘Motorcycles’”

```
db.products.aggregate([
  {
    "$match": {
      "msrp": { "$gt": 100.0 }
    }
  }
])
```

$gt, $gte, $lt, $lte are the other comparison operators.

Challenge #3: select only customers with a creditlimit under $50,000.

Challenge #4: select only customers that are not from USA.

Matching arrays

Some tricks for working with arrays.

```
db.customers.aggregate([
  {
    "$match": {
      "ordernumbers": 10123
    }
  }
])
```

Arrays can't equal individual values, so Mongo will interpret this as "contains": find the customer(s) whose ordernumbers array contains the value 10123.

```
db.customers.aggregate([
  {
    "$match": {
      "ordernumbers": { "$size" : 3 }
    }
  }
])
```

Find customers that made exactly 3 orders.

Combine the last two slides to find all customers with *more than* 3 orders.

```
db.customers.aggregate([
  {
    "$match": {
      "ordernumbers": { "$size" : {"$gt" : 3} }
    }
  }
])
```

Error: Expected a number in $size, got “{‘$gt’: 3}”.

Why? Because all these match conditions are short-hand for function calls.

`"msrp": { "$gt": 100.0 }` equivalent to `$gt(msrp, 100.0)` True if LHS > RHS.

`"productline": "Motorcycles"` equivalent to `$eq(productline, "Motorcycles")`

`"ordernumbers": { "$size" : 3 }` equivalent to `$eq($size(ordernumbers), 3)`

`"ordernumbers": { "$size" : {"$gt" : 3} }` equivalent to `$eq($size(ordernumbers), $gt(3))`

$gt(3) has no meaning.

Using $expr

Instead of matching a field, we can match an *expression*: any computation and comparison that we desire, using the $expr function as a trigger.

```
db.customers.aggregate([
  {
    "$match": {
      "$expr": {
        "$gt" : [{"$size" : "$ordernumbers"}, 3]
      }
    }
  }
])
```

equivalent to `$gt($size($ordernumbers), 3)`

When a field is an argument to a function call, we must put $ in front to mean “read the value of this field”; as opposed to using the literal name of the field as the argument.

Challenge #5: select only products that are low in stock (quantityinstock < 1000) and have a high msrp (msrp > 100).

Challenge #6: select only products that are in low in stock **OR** high msrp.

Combining stages

Since each pipeline stage outputs rows that can be used for another stage's input, we can **compose** (combine) them in a sequence.

```
db.products.aggregate([
  {
    "$match": {
      "productline": "Motorcycles"
    }
  },
  {
    "$project": {
      "productname": 1,
      "productline": 1,
      "_id": 0
    }
  }
])
```

Filter to Motorcycles, then project.

```
{
  productname: '1969 Harley Davidson Ultimate Chopper',
  productline: 'Motorcycles'
}
{
  productname: '1996 Moto Guzzi 1100i',
  productline: 'Motorcycles'
}
{
  productname: '2003 Harley-Davidson Eagle Drag Bike',
  productline: 'Motorcycles'
}
```

Pipeline stages: $sort

An easy pipeline function to sort the output.

```
db.customers.aggregate([
  {
    "$project": {
      "customername": 1
    }
  },
  {
    "$sort": {
      "customername": 1
    }
  }
])
```

```
{
  _id: 237,
  customername: 'ANG Resellers'
}
{
  _id: 187,
  customername: 'AV Stores, Co.'
}
{
  _id: 242,
  customername: 'Alpha Cognac'
}
{
  _id: 168,
  customername: 'American Souvenirs Inc'
}
{
  _id: 249,
```

Use -1 to sort in descending order.

Pipeline stages: $limit

An easy pipeline function to limit the output.

```
db.customers.aggregate([
  {
    "$project": {
      "customername": 1
    }
  },
  {
    "$limit": 3
  }
])
```

```
{
  _id: 103,
  customername: 'Atelier graphique'
}
{
  _id: 112,
  customername: 'Signal Gift Stores'
}
{
  _id: 114,
  customername: 'Australian Collectors, Co.'
}
```

Computing new fields

$project allows us to compute new values that aren't part of the input. (Same as using SELECT to make a computed column.) We simply define a **new** field in the projection, equal to an expression object.

```
db.employees.aggregate([
  {
    "$project": {
      "fullname" : {
        "$concat" : ["$firstname", " ", "$lastname"]
      }
    }
  }
])
```

```
{
  _id: 1002,
  fullname: 'Diane Murphy'
}
{
  _id: 1056,
  fullname: 'Mary Patterson'
}
{
  _id: 1076,
  fullname: 'Jeff Firrelli'
}
{
  _id: 1088,
  fullname: 'William Patterson'
```

The new field can be included with other fields to keep in the projection.

Example: select all Products that have at least a 90% profit margin on their suggested retail price. (Profit margin: (msrp - buyprice) / buyprice * 100.)

```
db.products.aggregate([
  {
    "$project": {
      "buyprice": 1,
      "msrp": 1,
      "margin": {
        "$multiply": [
          {
            "$divide": [{ "$subtract": ["$msrp", "$buyprice"] }, "$buyprice"]
          },
          100
        ]
      }
    }
  }
])
```

```
{
  _id: 'S10_1678',
  buyprice: 48.81,
  msrp: 95.7,
  margin: 96.06637984019667
}
{
  _id: 'S10_1949',
  buyprice: 98.58,
  msrp: 214.3,
  margin: 117.38689389328465
}
{
  _id: 'S10_2016',
  buyprice: 68.99,
  msrp: 118.94,
  margin: 72.40179736193653
}
{
  _id: 'S10_4698',
```

The same computation can be done in a $set stage instead of $project, if you want to keep all existing fields and also add the new one.

```
db.products.aggregate([
  {
    "$set": {
      "margin": {
        "$multiply": [
          {
            "$divide": [{ "$subtract": ["$msrp", "$buyprice"] }, "$buyprice"]
          },
          100
        ]
      }
    }
  }
])
```

```
{
  _id: 'S10_1678',
  productname: '1969 Harley Davidson Ultimate Chopper',
  productline: 'Motorcycles',
  productscale: '1:10',
  productvendor: 'Min Lin Diecast',
  productdescription: 'This replica features working kickstand, fr
  quantityinstock: 7933,
  buyprice: 48.81,
  msrp: 95.7,
  margin: 96.06637984019667
}
```

Conditional logic (akin to CASE ...WHEN...ELSE) can be done with $cond.

```
db.products.aggregate([
  {
    "$project": {
      "quantityinstock": 1,
      "stockstatus": {
        "$cond": [
          { "$gt": ["$quantityinstock", 5000] },
          "in stock",
          "low stock"
        ]
      }
    }
  }
])
```

```
{
  _id: 'S10_4757',
  quantityinstock: 3252,
  stockstatus: 'low stock'
}
{
  _id: 'S10_4962',
  quantityinstock: 6791,
  stockstatus: 'in stock'
}
{
  _id: 'S12_1099',
  quantityinstock: 68,
  stockstatus: 'low stock'
}
{
  _id: 'S12_1108',
  quantityinstock: 3619,
  stockstatus: 'low stock'
}
{
```

$cond takes three arguments:

1. The condition to test.
2. Value to use when the condition is true.
3. Value to use when the condition is false.

Challenge #7: list all customer names and whether they are "domestic" (USA) or "overseas" (all others).

Projecting with embedded children

We don’t need to use joins or groups in order to project, match, or aggregate fields from embedded children.

```
db.customers.aggregate([
  {
    "$project": {
      "customername": 1,
      "payments.amount" : 1
    }
  }
])
```

The . operator designates a field of an embedded object.

```
{
  _id: 103,
  customername: 'Atelier graphique',
  payments: [
    {
      amount: 6066.78
    },
    {
      amount: 14571.44
    },
    {
      amount: 1676.14
    }
  ]
}
{
  _id: 112,
  customername: 'Signal Gift Stores',
  payments: [
```

Matching with embedded children

We can also filter (match) based on the fields of an embedded child.

```
db.customers.aggregate([
  {
    "$match": {
      "payments.amount": { "$gt": 50000 }
    }
  },
  {
    "$project": {
      "customername": 1,
      "payments.amount": 1
    }
  }
])
```

This filter was “match any customer with a payment whose amount is > 50000.

```
{
  _id: 114,
  customername: 'Australian Collectors, Co.',
  payments: [
    {
      amount: 45864.03
    },
    {
      amount: 82261.22
    },
    {
      amount: 7565.08
    },
    {
      amount: 44894.74
    }
  ]
}
```

Aggregating with embedded children

We don't have to use groups in order to aggregate (min/max/sum/avg/count), if we are aggregating from a list embedded in the document.

```
db.customers.aggregate([
  {
    "$project": {
      "customername": 1,
      "totalpayments": {
        "$sum": "$payments.amount"
      }
    }
  }
])
```

The $sum function expects a list of values to add.

```
{
  _id: 103,
  customername: 'Atelier graphique',
  totalpayments: 22314.36
}
{
  _id: 112,
  customername: 'Signal Gift Stores',
  totalpayments: 80180.98
}
{
  _id: 114,
  customername: 'Australian Collectors, Co.',
  totalpayments: 180585.07
}
{
```

Match only the customers with at least $100,000 in payments.

```
db.customers.aggregate([
  {
    "$project": {
      "customername": 1,
      "totalpayments": {
        "$sum": "$payments.amount"
      }
    }
  },
  {
    "$match": {
      "totalpayments": { "$gte": 100000 }
    }
  }
])
```

There is no “HAVING” in MQL; you just use a $match after an aggregate field has been computed in a previous stage.

```
{
  _id: 114,
  customername: 'Australian Collectors, Co.',
  totalpayments: 180585.07
}
{
  _id: 119,
  customername: 'La Rochelle Gifts',
  totalpayments: 116949.68
}
{
  _id: 121,
  customername: 'Baane Mini Imports',
  totalpayments: 104224.79
}
{
  _id: 124,
```

Challenge #8: select the customer who made the largest payment.

Pipeline stages: $lookup

The equivalent to INNER JOIN. To join a child to their parent:

```
db.employees.aggregate([
  {
    "$lookup":{
      "from": "offices",
      "localField": "officecode",
      "foreignField": "_id",
      "as": "office"
    }
  },
  {
    "$project": {
      "firstname": 1,
      "lastname": 1,
      "officecode": 1,
      "office": 1
    }
  }
])
```

```
{
  _id: 1002,
  lastname: 'Murphy',
  firstname: 'Diane',
  officecode: '1',
  office: [
    {
      _id: '1',
      city: 'San Francisco',
      phone: '+1 650 219 4782',
      addressline1: '100 Market Street',
      addressline2: 'Suite 300',
      state: 'CA',
      country: 'USA',
      postalcode: '94080',
      territory: 'NA'
```

Mongo doesn't know that there can only be one Office with this officecode. To be safe, ALL join results are put in an **array** of values in the joined object, even if there can only be one foreign object.

$lookup from parent to children

When a parent has a list of child IDs, the lookup will match each ID to a child object.

```
db.customers.aggregate([
  {
    "$lookup": {
      "from": "orders",
      "localField": "ordernumbers",
      "foreignField": "_id",
      "as": "orders"
    }
  },
  {
    "$project": {
      "customername": 1,
      "orders._id": 1,
      "orders.orderdate": 1
    }
  }
])
```

```
{
  _id: 103,
  customername: 'Atelier graphique',
  orders: [
    {
      _id: 10345,
      orderdate: 2014-11-25T00:00:00.000Z
    },
    {
      _id: 10123,
      orderdate: 2013-05-20T00:00:00.000Z
    },
    {
      _id: 10298,
      orderdate: 2014-09-27T00:00:00.000Z
    }
  ]
}
{
  _id: 112,
```

Challenge #9: join customers to their (full) sales rep employee. Hint: the $lookup localField will be “salesrep.employeenumber”.

Pipeline stages: $group

To group documents together, in order to perform an aggregate function like max/sum.

```
db.employees.aggregate([
  {
    "$group": {
      "_id": "$jobtitle",
      "numEmployees": {"$count": {}}
    }
  },
  {
    "$sort": {
      "numEmployees": -1
    }
  }
])
```

```
{
  _id: 'Sales Rep',
  numEmployees: 17
}
{
  _id: 'VP Marketing'
  numEmployees: 1
}
{
  _id: 'VP Sales',
  numEmployees: 1
}
{
  _id: 'President',
  numEmployees: 1
}
```

The _id of the $group is the field to group by. Any other field in the $group is a projection to compute in terms of the grouped documents. Options:

{“$max”: “$fieldname”}

{“$min”: “$fieldname”}

{“$avg”: “$fieldname”}

{“$sum”: “$fieldname”}

{“$count”: {}}

Challenge #10: find the most expensive (msrp) product in each productline.

Challenge #11: count the number of “domestic” and “overseas” customers.

Ungrouping (?) documents

Embedded children can make some queries easier, like when we found the sum of all payments embedded in a customer. But what if we need to filter those children? As in, the equivalent of:

```
SELECT customernumber, SUM(amount)
FROM payments
WHERE amount > 10000
GROUP BY customernumber
```

Find the sum of each customer's payments that exceed $10000.

Why doesn't this work?

```
db.customers.aggregate([
  {
    "$match": {
      "payments.amount": { "$gt": 10000 }
    }
  },
  {
    "$project": {
      "totalpayments": {
        "$sum": "$payments.amount"
      }
    }
  }
])
```

Pipeline stages: $unwind

$unwind takes a document with an ar , producing one document *for each child* in the array. he parent object.

```
db.customers.aggregate([
  {
    "$unwind": {
      "path": "$payments",
    }
  },
  {
    "$project": {
      "customername": 1,
      "payments": 1,
    }
  }
])
```

```
{
  _id: 103,
  customername: 'Atelier graphique',
  payments: {
    checknumber: 'HQ336336',
    paymentdate: 2014-10-19T00:00:00.000Z,
    amount: 6066.78
  }
}
{
  _id: 103,
  customername: 'Atelier graphique',
  payments: {
    checknumber: 'JM555205',
    paymentdate: 2013-06-05T00:00:00.000Z,
    amount: 14571.44
  }
}
{
  _id: 103,
  customername: 'Atelier graphique',
  payments: {
    checknumber: 'OM314933'
```

Why is this helpful?

After unwinding, we can $match and $project each child document to meet our needs. If we need an aggregate on those children, we just group them back up.

```
db.customers.aggregate([
  {
    "$unwind": {
      "path": "$payments",
    }
  },
  {
    "$match": {
      "payments.amount": {"$gt": 10000}
    }
  },
  {
    "$group": {
      "_id": "$_id",
      "totalpayments": {
        "$sum": "$payments.amount"
      }
    }
  }
])
```

```
{
  _id: 103,
  totalpayments: 14571.44
}
{
  _id: 112,
  totalpayments: 80180.98
}
{
  _id: 114,
  totalpayments: 173019.99
}
{
```

Challenge #12: find the subtotal of each order.

Many-to-many associations

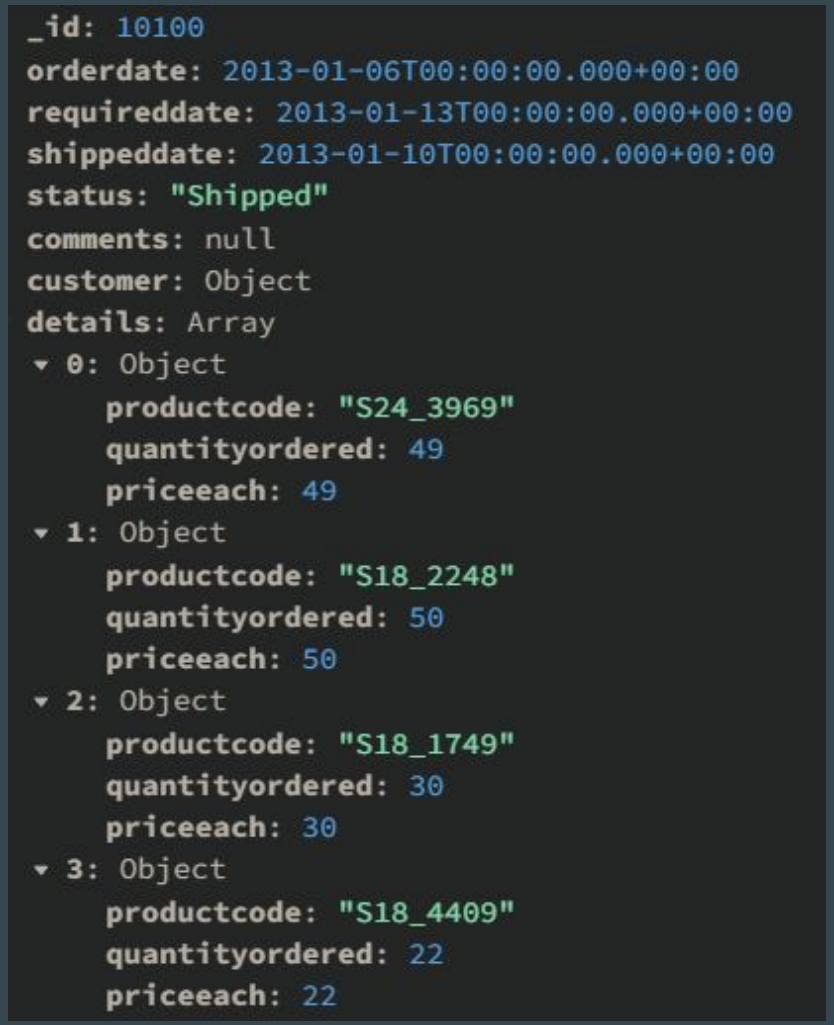
Many-to-many in MongoDB

The typical approach to a many-to-many in MongoDB is for both of the classes to embed an array of their related objects, as if they are both doing a “one to many”.

Those lists could store:

- Full copies of the related object, if there are only **a few**.
- The IDs of the related objects, if there are **many**.
- They should also store association attributes in the array, if there are any.

If there are too many associations to store in one object, a junction table can be used instead.



The many-to-many from Order to Product (via OrderDetails).

An Order only has a few OrderDetails, so we embed them directly. Each order detail has the quantityordered, priceeach, and productcode.

We don't *have* to embed OrderDetails in Products as well, if we can't imagine a *frequent need* to find all orders for a particular product. (And even then, we can always *index* the **orders.details.productcode** field to make this search go quickly.)

Many-to-many in JSON

Recall the three models we learned for many-to-many in JSON.

With embedded objects:

```
{
    "films": [
        {
            "title": "Top Gun",
            "releaseYear": 1986,
            "actors": [
                {
                    "role": "Maverick",
                    "actor": {
                        "name": "Tom Cruise"
                    }
                }
            ]
        }
```

```
"actors": [
        {
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "name": "Tom Cruise",
            "films": [
                {
                    "role": "Maverick",
                    "film": {
                        "title": "Top Gun",
                        "releaseYear": 1986
                    }
                }
            ]
        }
```

With embedded keys:


```
"films": [
    {
        "filmId": 1,
        "title": "Top Gun",
        "releaseYear": 1986,
        "actors": [
            {
                "role": "Maverick",
                "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476"
            }
        ]
    }
],
"actors": [
    {
        "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
        "name": "Tom Cruise",
        "films": [
            {
                "role": "Maverick",
                "filmId": 1
            }
        ]
    },
```

With junctions:


```
{
    "films": [
        {
            "filmId": 1,
            "title": "Top Gun",
            "releaseYear": 1986
        },
        {
            "filmId": 2,
            "title": "Mission: Impossible",
            "releaseYear": 1996
        }
    ],
    "actors": [
        {
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "name": "Tom Cruise"
        }
    ],
    "filmRoles": [
        {
            "filmId": 1,
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "role": "Maverick"
        },
        {
            "filmId": 2,
            "actorId": "2ac2f3d1-261f-42ad-8cea-5bb370bb8476",
            "role": "Ethan Hunt"
        },
```

Many-to-many in MongoDB

We generally don't embed full objects in MongoDB many-to-many associations; the choice is between embedded *keys* and junction tables. Like one-to-many vs. one-to-squillions, the choice is often *how many A's and B's are there*?

- Many: if each A and B can fit an array of the IDs of their related objects, then use embedded keys.

- Squillions: if there are more A-B pairings than can fit into a document of A or of B, then use a junction.

Denormalization

Redundancy and denormalization

Document databases are more comfortable with redundancy than relational databases. Redundant copies of data can save you from having to do a join, at the small expense of a few more bytes of storage.

If **normalization** is the process of eliminating redundancy by extracting attributes into new classes, ***denormalization*** is the the process of *introducing* redundancy by copying or flattening data from one collection to another.

Why would we do this? To avoid needing a join! If there is some common operation in our application that requires data from an object along with a related object, *denormalizing* the related object's fields can avoid a join for this common operation.

```
// cookietypes collection
{
    "_id" : "...",
    "name" : "Thin Mints",
    "price" : 6.0,
    "ingredients" : [1, 4, 13]
}
{
    "_id" : "...",
    "name" : "Samoas",
    "price" : 6.0,
    "ingredients" : [2, 1]
}

// ingredients collection
{
    "_id" : 1,
    "name" : "sugar"
}
{
    "_id" : 2,
    "name" : "coconut"
}
{
    "_id" : 4,
    "name" : "cocoa"
}
{
    "_id" : 13,
    "name" : "milk"
}
```

No redundancy, requires a join.

```
// cookietypes collection
{
    "_id" : "...",
    "name" : "Thin Mints",
    "price" : 6.0,
    "ingredients" : ["sugar","cocoa","milk"]
}
{
    "_id" : "...",
    "name" : "Samoas",
    "price" : 6.0,
    "ingredients" : ["coconut","sugar"]
}
```

Redundancy, no join required.

Is the time it takes to perform a join, plus the 4 bytes per ID stored in the ingredients list, worth it to save the bytes needed for each ingredient? Maybe not...

```
// cookietypes collection
{
    "_id" : "...",
    "name" : "Thin Mints",
    "price" : 6.0,
    "ingredients" : ["sugar","cocoa","milk"]
}
{
    "_id" : "...",
    "name" : "Samoas",
    "price" : 6.0,
    "ingredients" : ["coconut","sugar"]
}
```

Potential problem: how do we find all cookies that have sugar?

```
db.cookietypes.aggregate([
  {
    "$match": {
      "ingredients": "sugar"
    }
  }
])
```

Row scan! Uh oh... How can we be more efficient?

Problems with denormalization

Denormalization introduces redundancy, which we have noted has problems:

1. Extra storage costs for the redundant copies. (Each showing now copies the film's title and room name.)

2. To update a film's title, we must also update every showing that references it. (A single document can be updated atomically. Multiple documents cannot.)

When to consider denormalization

There are no algorithmic rules for denormalization. We must weigh the pros and cons in a given context, as engineers!

- Redundancy *generally* trades *space* (extra copies) to gain *time* (fewer joins). Which is more valuable in your context?
  - Think about the queries you will run most frequently in your application. Is there a way to reduce the number of joins with a small amount of redundancy? Then consider doing that!
  - The most common use for a “showing” object is to display to a user. Users want to know the film of that showing, and where it is. To avoid a join, we can denormalize those fields into the showing objects.

- Redundancy requires multiple updates when objects change. How often do those objects change in your context?
  - How often does a film’s title or room name need to change *after* showings have been scheduled?