MONGODB

a NoSQL, document-oriented database

DATABASES

organized collections of data

Database Models

RELATIONAL, SQL

1970s

split data into a series of **normalized** tables

use **joins** to combine data in different tables together

en.wikipedia.org/wiki/Database

Not only Sql

NoSQL

2000s

fast key-value stores and document-oriented databases (JSON, XML)

do not require fixed table schemas, no support for joins

scale horizontally

en.wikipedia.org/wiki/Database

Mongo

Databases

Tables

Rows

MongoDB

Databases

Tables

Collections

Rows

Documents

MONGO DOCUMENTS

documents are JSON-like

stored as BSON

documents must be smaller than 16MB

Both of these documents can be stored in the same collection

```
{"type":"llama", height:1.8}
{"type":"camel", height:2.2, humps:2}
```

Why have separate collections?

developers aren’t confused

query efficiency

data locality

indexing (defined per collections)

Data Types

BASIC TYPES

JSON: null, boolean, number, string, array, and object

MongoDB: null, boolean, number, string, array, **date**, **regex**, **embedded document**, **object id**, **binary data**, **code**

EMBEDDED DOCUMENTS

```
{
    "type": "llama",
    "name": "Francesca",
    "height": 1.8,
    "farm": {
      "name": "Silver Lake",
      "owner": "Goldilocks"
    }
}
```

OBJECTIDS

Every document must have an **`"_id"`** key

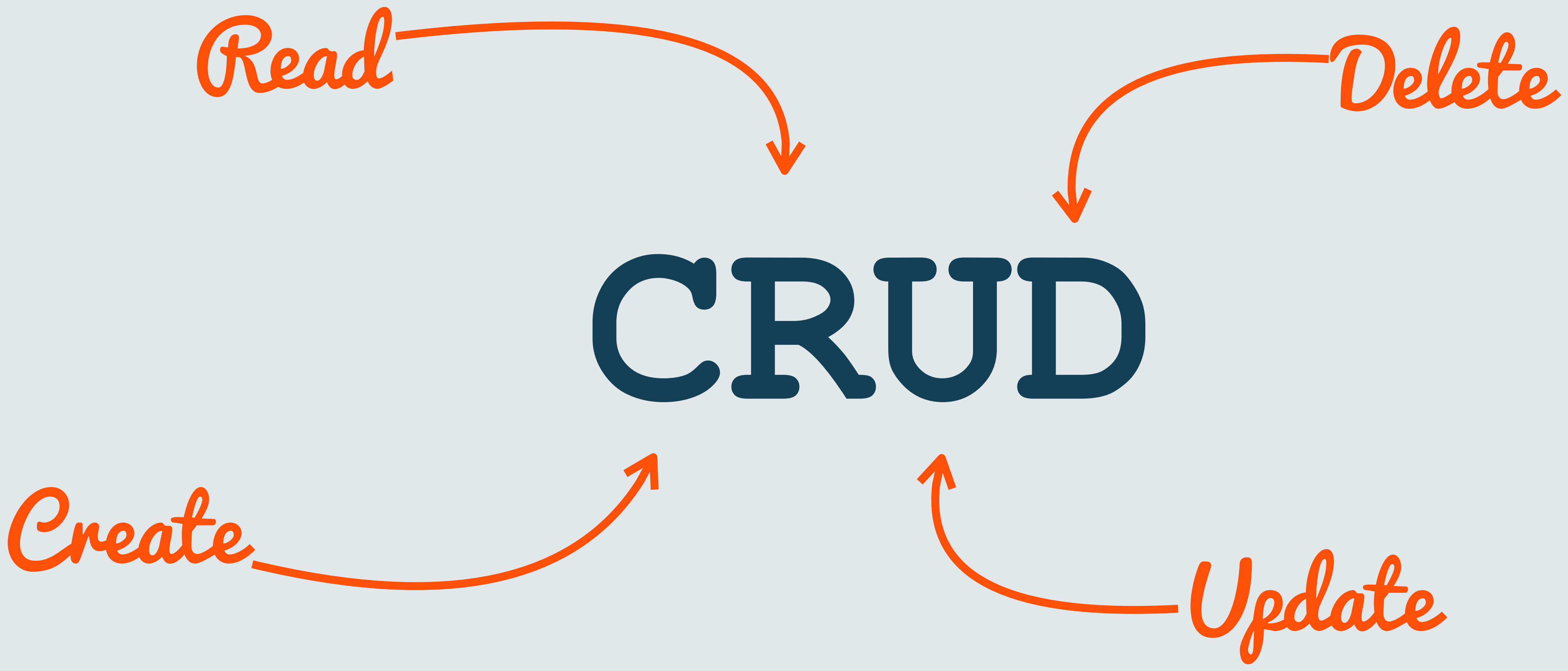
Every document in a collection must have a unique **`"_id"`** key

ObjectId is the default type for **`"_id"`**

OBJECTIDS

0-3	4-6	7-8	9-11
Timestamp	Machine	PID	Increment

~17M unique ObjectIds per process per second



Read

Delete

CRUD

Create

Update

Create

```
> llama = {
    "type": "llama",
    "name": "Francesca",
    "height": 1.8,
    "date" : new Date()
}

> db.camelids.insert(llama);
```

Read

```
> db.camelids.findOne()

{
   "_id" : ObjectId("54fda10dd452eebae749a0b8"),
   "type" : "llama",
   "name" : "Francesca",
   "height" : 1.8,
   "date" : ISODate("2015-03-09T13:32:43.737Z")
}
```

Update

```
> llama.diet = ["grass","hay"]

> db.camelids.update({"type" :
"llama"}, llama)
```

Update

```
> db.camelids.findOne()

{
   "_id" : ObjectId("54fda10dd452eebae749a0b8"),
   "type" : "llama",
   "name" : "Francesca",
   "height" : 1.8,
   "date" : ISODate("2015-03-09T13:32:43.737Z"),
   "diet" : [
      "grass",
      "hay"
   ]
}
```

Delete

```
> db.camelids.remove()
> db.camelids.remove({type : “llama"})
> db.camelids.drop()
```

Update Modifiers

Update

```
> llama.diet = ["grass","hay"]

> db.camelids.update({"type" : "llama"},
llama)

> db.camelids.update({"type" : "llama"},
{"name": "maria"})
```

What does this do?

Update

```
> db.camelids.findOne()

{ "_id" : ObjectId("54fda10dd452eebae749a0b8"),
"name" : "maria" }
```

Use **update modifiers** to update portions of a document

```
> db.camelids.update({"type" : "llama"},
{"$set": {"name": “maria”}})

> db.camelids.findOne()
{
   "_id" : ObjectId("54fda779d452eebae749a0ba"),
   "date" : ISODate("2015-03-09T14:00:22.530Z"),
   "height" : 1.8,
   "name" : "maria",
   "type" : "llama"
}
```

Use **update modifiers** to update portions of a document

```
> db.camelids.update({"type" : "llama"},
{"$inc": {"height": 0.2}})
```

```
> db.camelids.findOne()
{
   "_id" : ObjectId("54fda779d452eebae749a0ba"),
   "date" : ISODate("2015-03-09T14:00:22.530Z"),
   "height" : 2,
   "name" : "maria",
   "type" : "llama"
}
```

Array Modifiers

$push, $pop, $pull

$each

$sort, $slice

$ne/$push, $addToSet/$each

positional access

Queries and $-Conditionals

QUERYING IN MONGO

find() & findOne()

$-conditionals

queries return db cursor that lazily returns batches of documents

Basic Queries

```
> db.camelids.find()
> db.camelids.find({"type" : "llama"})
> db.camelids.find({"type" : "llama","name" :
"Francesca"})
{
   "_id" : ObjectId("54fda10dd452eebae749a0b8"),
   "type" : "llama",
   "name" : "Francesca",
   "height" : 1.8,
   "date" : ISODate("2015-03-09T13:32:43.737Z")
}
```

Specify which keys to return

```
> db.camelids.findOne({“type":"llama"},
{"_id":0,"name":1})
```

```
{ "name" : "maria" }
```

$-Conditionals

```
> db.camelids.findOne({"height":{"$lte"
: 1.5, "$gte" : 1.2}})

> db.camelids.findOne({"type" :
{"$in" : [“llama”,”alpaca"]}})

> db.camelids.find({"$or" : [{"type" :
"alpaca"},{"name" : "Francesca"}])
```

Schema Design

ONE-TO-FEW

```
> db.person.findOne()
{
  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' }
  ]
}
```

embedded document

blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1

ONE-TO-MANY

```
> db.parts.findOne()
{
    _id : ObjectID('AAAA'),
    partno : '123-aff-456',
    name : '#4 grommet',
    qty: 94,
    cost: 0.94,
    price: 3.99
}
```

each **`part`** has own document

blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1

ONE-TO-MANY

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [
        ObjectID('AAAA'),
        ObjectID('F17C'),
        ObjectID('D2AA'),
        // etc
    ]
```

array of references to `part` documents

blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1

ONE-TO-MANY

```
> product =
db.products.findOne({catalog_number: 1234});

> product_parts = db.parts.find({_id:
{ $in : product.parts } } ).toArray() ;
```

application-level join

blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1

ONE-TO-GAZILLION

```
> db.hosts.findOne()
{
    _id : ObjectID('AAAB'),
    name : 'goofy.example.com',
    ipaddr : '127.66.66.66'
}

> db.logmsg.findOne()
{
    time : ISODate("2014-03-28T09:42:41.382Z"),
    message : 'cpu is on fire!',
    host: ObjectID('AAAB')
}
```

parent-referencing

blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1

ONE-TO-GAZILLION

```
> host = db.hosts.findOne({ipaddr :
'127.66.66.66'});

> last_5k_msg = db.logmsg.find({host:
host._id}).sort({time :
-1}).limit(5000).toArray()
```

application-level join

blog.mongodb.org/post/87200945828/6-rules-of-thumb-for-mongodb-schema-design-part-1

TWO-WAY REFERENCING

```
db.person.findOne()
{
    _id: ObjectID("AAF1"),
    name: "Kate Monster",
    tasks [
        ObjectID("ADF9"),
        ObjectID("AE02"),
        ObjectID("AE73")
        // etc
    ]
}
```

array of references to **`task`** documents

blog.mongodb.org/post/87892923503/6-rules-of-thumb-for-mongodb-schema-design-part-2

TWO-WAY REFERENCING

```
db.tasks.findOne()
{
    _id: ObjectID("ADF9"),
    description: "Write lesson plan",
    due_date:  ISODate("2014-04-01"),
    owner: ObjectID("AAF1")
}
```

reference to **`person`** document

blog.mongodb.org/post/87892923503/6-rules-of-thumb-for-mongodb-schema-design-part-2

DENORMALIZING MANY-TO-ONE

```
> db.products.findOne()
{
    name : 'left-handed smoke shifter',
    manufacturer : 'Acme Corp',
    catalog_number: 1234,
    parts : [
        { id : ObjectID('AAAA'), name : '#4 grommet' },
        { id: ObjectID('F17C'), name : 'fan blade assembly' },
        { id: ObjectID('D2AA'), name : 'power switch' },
        // etc
    ]
}
```

no join required to list **`part`** names

blog.mongodb.org/post/87892923503/6-rules-of-thumb-for-mongodb-schema-design-part-2

DENORMALIZING MANY-TO-ONE

```
> product = db.products.findOne({catalog_number:
1234});
```

```
> part_ids = product.parts.map( function(doc)
{ return doc.id } );
```

a little more work to application-level join

```
> product_parts = db.parts.find({_id: { $in :
part_ids } } ).toArray() ;
```

blog.mongodb.org/post/87892923503/6-rules-of-thumb-for-mongodb-schema-design-part-2

STRUCTURING DATA

For “one-to-few”, you can use an array of embedded documents

For “one-to-many”, or on occasions when the “N” side must stand alone, you should use an array of references. You can also use a “parent-reference” on the “N” side if it optimizes your data access pattern

For “one-to-squillions”, you should use a “parent-reference” in the document storing the “N” side

blog.mongodb.org/post/88473035333/6-rules-of-thumb-for-mongodb-schema-design-part-3

CONSIDERATIONS

What is the cardinality of the relationship: is it “one-to-few”, “one-to-many”, or “one-to-squillions”?

Do you need to access the object on the “N” side separately, or only in the context of the parent object?

What is the ratio of updates to reads for a particular field?

blog.mongodb.org/post/88473035333/6-rules-of-thumb-for-mongodb-schema-design-part-3

RULES OF THUMB

favor embedding unless there is a compelling reason not to

needing to access an object on its own is a compelling reason not to embed it

high-cardinality arrays are a compelling reason not to embed

blog.mongodb.org/post/88473035333/6-rules-of-thumb-for-mongodb-schema-design-part-3

RULES OF THUMB

if you index correctly and use the projection specifier, application-level joins are barely more expensive than server-side joins in a relational database.

consider the write/read ratio when denormalizing

model your data according to application's data access patterns

blog.mongodb.org/post/88473035333/6-rules-of-thumb-for-mongodb-schema-design-part-3

NEXT CLASS: NodeJS