Modeling data for NoSQL

the MongoDB (document store) method

Theme #1:

Great Information Architecture involves much more than the database

- Easily understood structures
- Harmonized with software
- Acknowledging impedance issues

Theme #2:
Today's solutions need to accommodate tomorrow's needs

- End of "Requirements Complete"
- Ability to rapidly pivot
- Shorter solutions lifecycles

~~Theme #n:~~

Pipe to dev/null = webscale!1!
(but does it support sharding?)

Terminology

RDBMS		Document database (MongoDB)
Database	→	Database
Table	→	Collection
Index	→	Index
Row	→	Document
Column	→	Field
Join	→	Embedding, Linking, $lookup

Why Documents? Developer Agility

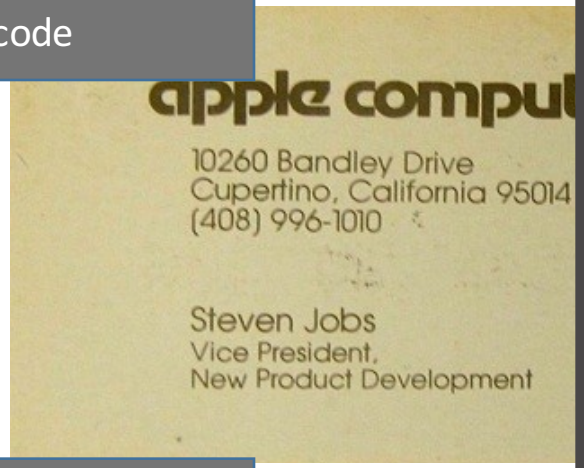
Drivers	Features	Use Cases
Simplicity: Data fits documents better than tabular relations enabling greater developer productivity	Semi-structure (JSON) Low Object Impedance (no ORM needed)	**CMS**: eg) news articles **EHR:** eg) patient records **Web/Mobile Apps:** End-to-end JSON
Dynamic Schema: schema too volatile and variable to practically maintain in a relational model.	Support for multiple schemas Easy manipulation of attributes	**SaaS:** no downtime (lazy) schema migrations **Data Hub:** easy integration and MDM (eg. Customer) across a large number of external systems. **Product Catalog:** volatile schema and many unknown attributes.

Modeling for Simplicity

Contact
- name
- company
- title

Address
- type
- street
- city
- state
- zip_code

Phone
- type
- number



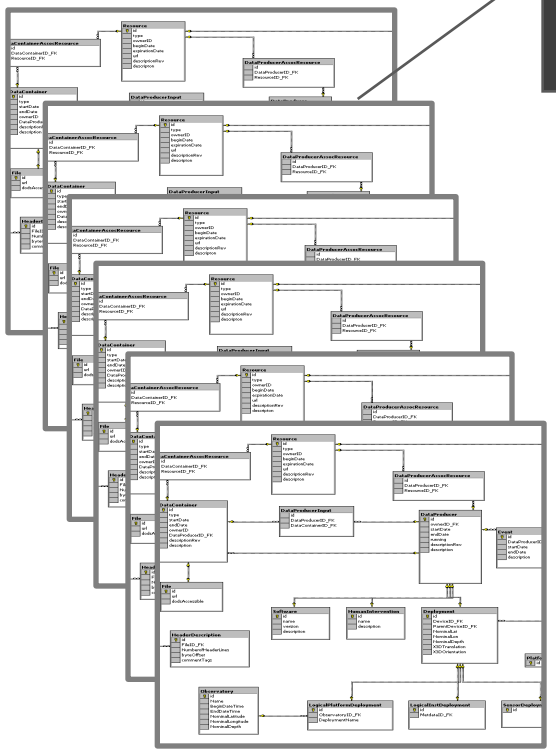
Embedded Hierarchal Elements

```
{
  "_id": 2,
  "name": "Steven Jobs",
  "title": "VP, New Product Development",
  "company": "Apple Computer",
  "address": {
      "street": "10260 Bandley Dr",
      "city": "Cupertino",
      "state": "CA",
      "zip_code": 95014,
      "country": "USA"
  },
  "phone":[{"type": "land", "n":"408-996-1010"},
           {"type": "cell", "n":"408-996-1111"}]
}
```

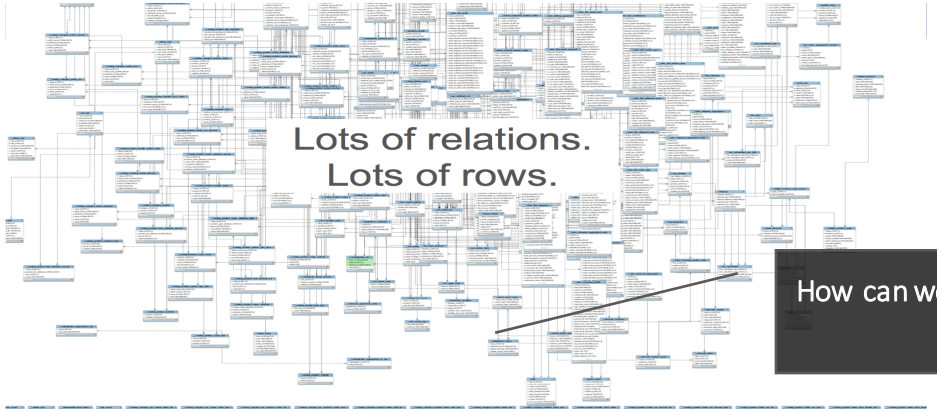
List Structures

19 Supported Data Types (bsonspec.org)

Relational Model Challenges

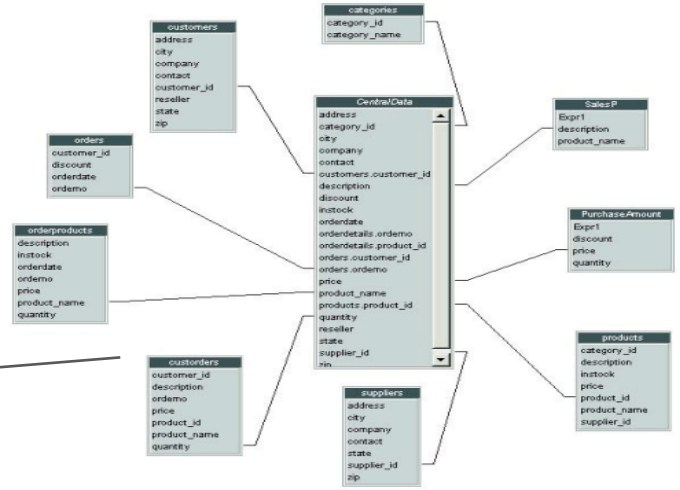


50+ Different Schemas

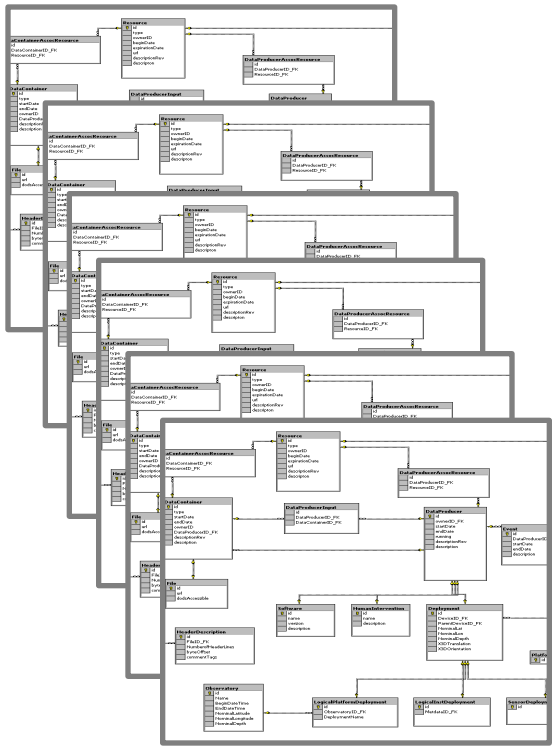


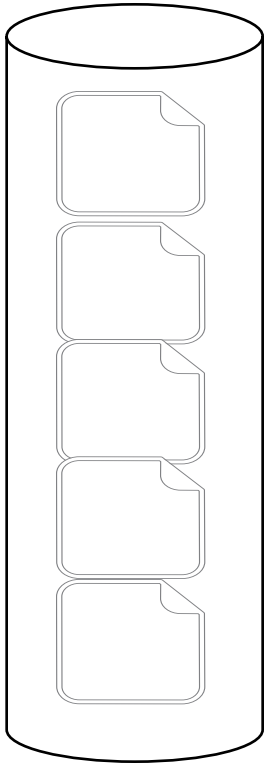
How can we translate this into a Data Hub?

ETL 50+ applications into a Dimensional Model? Integrating a few is hard…



Strategy: Leverage Dynamic Schema to Support a Variety of Data Models






```
db.books.find(
{
        "author.first" : "Stephen",
        "author.last" : "King",
        "type" : {
                $in["Fiction","Horror","Sci-Fi"]
                }
})
```

Collection of Books

What is a Document?

```
{
     _id: “123”,
    title: "MongoDB: The Definitive Guide",
    authors: [
        { _id: "kchodorow", name: "Kristina Chodorow“ },
        { _id: "mdirold", name: “Mike Dirolf“ }
    ],
    published_date: ISODate(”2010-09-24”),
    pages: 216,
    language: "English",
    thumbnail: BinData(0,"AREhMQ=="),
    publisher: {
        name: "O’Reilly Media",
        founded: 1980,
        locations: ["CA”, ”NY” ]
    }
}
```

Documents Map to Language Constructs

```
// Java: maps
DBObject query = new
BasicDBObject("publisher.founded", 1980));
Map m = collection.findOne(query);
Date pubDate = (Date)m.get("published_date"); //
java.util.Date


// Javascript: objects
m = collection.findOne({"publisher.founded" :
1980});
pubDate = m.published_date;    // ISODate
year = pubDate.getUTCFullYear();


# Python: dictionaries
m = coll.find_one({"publisher.founded" : 1980 });
pubDate = m["pubDate"].year   # datetime.datetime
```

Traditional Data Design

- Static, Uniform Scalar Data
- Rectangles
- Low-level, physical representation

Document Data Design

- Flexible, Rich Shapes
- Objects
- Higher-level, business representation

Schema Design By Example

Library Management Application

- Patrons/Users
- Books
- Authors
- Publishers

Question:
What is a Patron’s Address?

Patron + Address: Initial Attempt

```
> db.patrons.find({ _id : “joe” })
{
    _id: "joe“,
    name: "Joe Bookreader”,
    favoriteGenres: [ ”mystery”, ”programming” ]
}


> db.addresses.find({ _id : “joe” })
{
    _id: "joe“,
    street: "123 Fake St.",
    city: "Faketon",
    state: "MA",
    zip: “12345”
}
```

Patron + Address: The MongoDB Way

```
> db.patrons.find({ _id : “joe” })
{
    _id: "joe",
    name: "Joe Bookreader",
    favoriteGenres: [ ”mystery”, ”programming” ]
    address: {
        street: "123 Fake St. ",
        city: "Faketon",
        state: "MA",
        zip: “12345”
    }
}
```

Projection: Return only what you need

```
> db.patrons.find({ _id : “joe” }, {“_id”: 0,
”address”:1})
{
    address: {
        street: "123 Fake St. ",
        city: "Faketon",
        state: "MA",
        zip: “12345”
    }
}
> db.patrons.find({ _id : “joe” }, {“_id”: 0,
“name”:1,    ”address.state”:1})
{
    name: "Joe Bookreader",
    address: {
        state: "MA”
    }
}
```

Substructure Works Well With Code

```
> addr = db.patrons.find({_id :“joe”},{“_id”: 0,”address”:1})
{
    address: {
        street: "123 Fake St. ",
        city: "Faketon",
        state: "MA",
        zip: “12345”
    }
}

// Pass the whole Map to this function:
doSomethingWithOneAddress(addr);

// Somewhere else in the code is the actual function:
doSomethingWithOneAddress(Map addr)
{ // Look for state }
```

Remember: Document Shapes Can Vary

```
> db.patrons.insert({ _id : “bob”,
    name: ”Bob Nobooks",
    address: {
        street: ”139 W45 St. ",
        city: ”NY",
        state: ”NY",
        country: ”USA”
    }
}
> db.patrons.find({},{“_id”: 1, ”address”:1})
{   _id: “joe”,
    address: {
        street: "123 Fake St. ",
        city: "Faketon",
        state: "MA",
        zip: “12345”
    }
}
{   _id: ”bob",
    address: {
        street: ”139 W45 St. ",
        city: ”NY",
        state: ”NY",
        country: ”USA”
    }
}
```

Substructure Amplifies Agility

```
> addr = db.patrons.find({_id :"bob"},{"_id": 0,"address":1})
{
    address: {
        street: "139 W45 St. ",
        city: "NY",
        state: "NY",
        country: "USA"
    }
}


// Pass the whole Map to this function:
doSomethingWithOneAddress(addr);


doSomethingWithOneAddress(Map addr)
{ // Look for state and optional country }
```

NO CHANGE to queries

NO COMPILE-TIME DEPENDENCIES when passing Maps

Only the single implementation that looks for country needs to change

The Advantage over Rectangles

```
resultSet = select street, state, city, country, …
```

Queries must change to pick up new columns

```
Map addr = processIntoMap(resultSet);

// Pass the whole Map to this function:
doSomethingWithOneAddress(addr);


doSomethingWithOneAddress(Map addr)
{ // Look for state and optional country }
```

Compile-time dependency to process new columns to Map

Substructure Scales For Change

MongoDB

```
db.patrons.find({},
{"myAddress":1,"yourAddress":1,"brokerAddress":1,
"momsAddress":1, …})
```

Traditional SQL

```
resultSet = select mystreet, mystate, mycity, mycountry,
yourstreet, yourstate, yourcity, yourcountry, brokerstreet,
brokerstate, brokercity, brokercountry, momsstreet,
momsstate, momscity, momscountry, …
```

One-to-One Relationships

- “Belongs to” relationships are often embedded
- Holistic representation of entities with their embedded attributes and relationships
- Great read performance

Most important:

- **Keeps simple things simple**
- **Frees up time to tackle harder schema design issues**

Question:

What are a Patron's Addresses?

A Patron and their Addresses

```
> db.patrons.find({ _id : “bob” })
{
    _id: “bob",
    name: “Bob Knowitall",
    addresses: [
      {street: "1 Vernon St.", city: "Newton", …},
      {street: "52 Main St.", city: "Boston", …}
    ]
}
```

A Patron and their Addresses

```
> db.patrons.find({ _id : “bob” })
{
    _id: “bob",
    name: “Bob Knowitall",
    addresses: [
      {street: "1 Vernon St.", city: "Newton", …},
      {street: "52 Main St.", city: "Boston", …}
    ]
}

> db.patrons.find({ _id : “joe” })
{
    _id: "joe",
    name: "Joe Bookreader",
    address: { street: "123 Fake St. ",  city: "Faketon", …}
}
```

Migration Options

- Migrate all documents when the schema changes.
- Migrate On-Demand
  - As we pull up a patron's document, we make the change.
  - Any patrons that never come into the library never get updated.
- Leave it alone
  - The code layer knows about both address and addresses

Letting The Code Deal With Documents

```
Map d = collection.find(new BasicDBObject("_id","bob"));

// Contract: Return either a List of addresses or a null
// if no addresses exist

// Try to get the new "version 2" shape:
List addrl = (List) d.get("addresses");

// If not there, try to get the old one:
if(addrl == null) {
  Map oneAddr = (Map) d.get("address");
  if(oneAddr != null) {
    addrl = new List();
    addrl.append(oneAddr);
  }
}

// addrl either exists with 1 or more items or is null
```

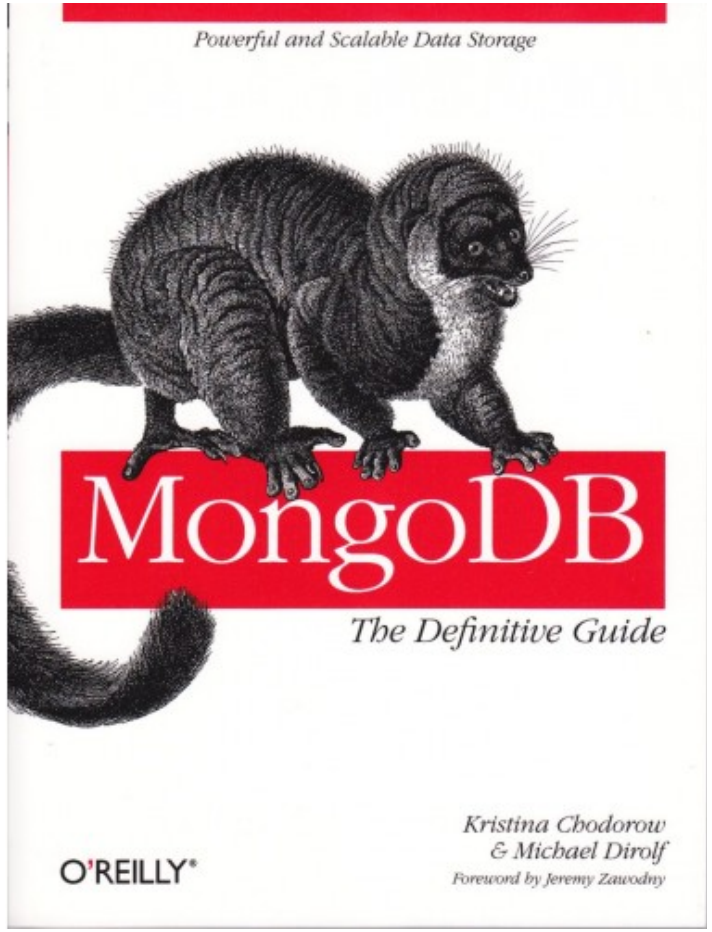
Question:

Who is the publisher of this book?

Book

- MongoDB: The Definitive Guide,
- By Kristina Chodorow and Mike Dirolf
- Published: 9/24/2010
- Pages: 216
- Language: English

- Publisher: O'Reilly Media, CA



Book with Embedded Publisher

```
> book = db.books.find({ _id : “123” })
{
    _id: “123”,
    title: "MongoDB: The Definitive Guide",
    authors: [ "Kristina Chodorow", "Mike Dirolf" ],
    published_date: ISODate(”2010-09-24”),
    pages: 216,
    language: "English",
    publisher: {
        name: "O’Reilly Media",
        founded: 1980,
        locations: ["CA”, ”NY” ]
    }
}
```

Better: Don’t Forget the Substructure!

```
> book = db.books.find({ _id : “123” })
{
    _id: “123”,
    title: "MongoDB: The Definitive Guide",
    authors: [
        { first: "Kristina”, last: “Chodorow” },
        { first: ”Mike”, last: “Dirolf” }
    ],
    published_date: ISODate(”2010-09-24”),
    pages: 216,
    language: "English",
    publisher: {
        name: "O’Reilly Media",
        founded: 1980,
        locations: ["CA”, ”NY” ]
    }
}
```

One-To-Many Using Embedding

- Optimized for read performance of Books
- We accept data duplication
- An index on “publisher.name” provides:
  - Efficient lookup of all books for given publisher name
  - Efficient way to find all publisher names (distinct)
- Does not automatically mean there is no “master” Publisher collection (from which data is copied when creating a new Book)

Publishers as a Separate Entity

```
> publishers = db.publishers.find()
{
    _id: “oreilly”,
    name: "O’Reilly Media",
    founded: 1980,
    locations: ["CA”, ”NY” ]
}
{
    _id: “penguin”,
    name: ”Penguin”,
    founded: 1983,
    locations: [ ”IL” ]
}
```

Single Book with Linked Publisher

```
> book = db.books.find({ _id: “123” })
{
    _id: “123”,
    publisher_id: “oreilly”,
    title: "MongoDB: The Definitive Guide",
    …
}

> db.publishers.find({ _id : book.publisher_id })
{
    _id: “oreilly”,
    name: "O’Reilly Media",
    founded: 1980,
    locations: ["CA”, ”NY” ]
}
```

Multiple Books with Linked Publisher

```
db.books.find({ pages: {$gt:100}} ).forEach(function(book) {
    // Do whatever you need with the book document, but
    // in addition, capture publisher ID uniquely by
    // using it as a key in an object (Map)
    tmpm[book.publisher.name] = true;
    });

uniqueIDs = Object.keys(tmpm); // extract ONLY keys

db.publishers.find({"_id": {"$in": uniqueIDs } });
```

The Basic MongoDB Application Side "Join"

Using $lookup

```
{
      $lookup:
            {
                  from: <collection to join>,
                  localField: <field from the input documents>,
                  foreignField: <field from the documents of the "from" collection>,
      as: <output array field>
             }
}

db.books.aggregate([{$lookup:{from:”publishers",localField:”_id",foreignField:”publisher_id",as:”books_by_pub"}}])

{
     _id: “123”,
     publisher_id: “oreilly”,
     title: "MongoDB: The Definitive Guide",
     …,
     books_by_pub: [{
     _id: “oreilly”,
     name: "O’Reilly Media",
     founded: 1980,
     locations: ["CA”, ”NY” ]
     },…]
}
```

Cartesian Product != Desired Structure

```
resultSet = “select B.name, B.publish_date, P.name, P.founded
            from Books B, Publisher P
            where P.name = B.publisher_name
            and B.pages > 100”
```

B.Name	B.publish_date	P.name	P.founded
More Jokes	2003	Random House	1843
Perl Tricks	1998	O’Reilly	1980
More Perl	2000	O’Reilly	1980
Starting Perl	1996	O’Reilly	1980
Flying Kites	1980	Random House	1843
Using Perl	2002	O’Reilly	1980
Bad Food	2011	Random House	1843

…And Tough To Use Without ORDER BY

```
resultSet = "select B.name, B.publish_date, P.name, P.founded
             from Books B, Publisher P
             where P.name = B.publisher_name
             and B.pages > 100
             order by P.name";
```

B.Name	B.publish_date	P.name	P.founded
Perl Tricks	1998	O'Reilly	1980
More Perl	2000	O'Reilly	1980
Using Perl	2002	O'Reilly	1980
Starting Perl	1996	O'Reilly	1980
Flying Kites	1980	Random House	1843
Bad Food	2011	Random House	1843
More Jokes	2003	Random House	1843

SQL Is About Disassembly

```
resultSet = "select B.name, B.publish_date, P.name, P.founded
             from Books B, Publisher P
             where P.name = B.publisher_name
             and B.pages > 100
             order by P.name";

prev_name = null;
while(resultSet.next()) {
    if(!resultSet.getString("P.name").equals(prevName)) {
       // "next" publisher name found.  Process material
       // accumulated and reset for next items.
       makeNewObjects();   //etc.
       prev_name = resultSet.getString("P.name")
    }
}
```

One-To-Many Using Linking

- Optimized for efficient management of mutable data
- Familiar way to organize basic entities
- Code is used to assemble fetched material into other objects, not disassemble a single ResultSet
  - More complicated queries may be easier to code and maintain with assembly vs. disassembly

Question:

Who are the authors of a given book?

Books with linked Authors

```
> book = db.books.find({ _id : “123” })
{
    _id: “123”,
    title: "MongoDB: The Definitive Guide",
    …
    authors: [
        { _id: “X12”, first: "Kristina”, last: “Chodorow” },
        { _id: “Y45”, first: ”Mike”, last: “Dirolf” }
    ],
}

> a2 = book.authors.map(function(r) { return r._id; });
> authors = db.authors.find({ _id : { $in : a2}})

{_id:”X12”,name:{first:"Kristina”,last:”Chodorow”},hometown: …
}
{_id:“Y45”,name:{first:”Mike”,last:”Dirolf”}, hometown: … }
```

Question:

What are all the books an author has written?

Double Link Books and Authors

```
> db.authors.find({ _id : “X12” })
{
    _id: ”X12",
    name: { first: "Kristina”, last: “Chodorow” } ,
    hometown: "Cincinnati",
    books: [ {id: “123”, title : "MongoDB: The Definitive
Guide“ } ]
}
> db.books.find({ _id : “123” })
{
    _id: “123”,
    title: "MongoDB: The Definitive Guide",
    …
    authors: [
        { _id: “X12”, first: "Kristina”, last: “Chodorow” },
        { _id: “Y45”, first: ”Mike”, last: “Dirolf” }
    ],
}
```

Another Approach: Index The Array

```
> db.books.find({ _id : “123” })
{
    authors: [
        { _id: “X12”, first: "Kristina”, last: “Chodorow” },
        { _id: “Y45”, first: ”Mike”, last: “Dirolf” }
    ],
}

> db.books.createIndex({“authors._id”: 1});

> db.books.find({ “authors._id” : “X12” }).explain();
{
    "cursor" : "BtreeCursor authors.id_1",
…
    "millis" : 0,
}
```

Embedding vs. Linking

- **Embedding**
  - Terrific for read performance
    - Webapp “front pages” and pre-aggregated material
    - Complex structures
  - Great for insert-only / immutable designs
  - Inserts might be slower than linking
  - Data integrity for not-belongs-to needs to be managed

- **Linking**
  - Flexible
  - Data integrity is built-in
  - Work is done during reads
    - But not necessarily more work than RDBMS

Question:

What are the personalized attributes for each author?

Assign Dynamic Structure to a Known Name

```
> db.authors.find()
{
    _id: ”X12",
    name: { first: "Kristina”, last: “Chodorow” },
    personalData: {
        favoritePets: [ “bird”, “dog” ],
        awards: [ {name: “Hugo”, when: 1983}, {name: “SSFX”,
when: 1992} ]
    }
}
{
    _id: ”Y45",
    name: { first: ”Mike”, last: “Dirolf” } ,
    personalData: {
        dob: ISODate(“1970-04-05”)
    }
}
```

A Fundamental And Powerful Document Model Feature!

Polymorphism: Worth an Extra Slide

```
> db.events.find()
{ type: ”click", ts: ISODate(“2015-03-03T12:34:56.789Z”,
data: { x: 123, y: 625, adId: “AE23A” } }

{ type: ”click", ts: ISODate(“2015-03-03T12:35:01.003Z”,
data: { x: 456, y: 611, adId: “FA213” } }

{ type: ”view", ts: ISODate(“2015-03-03T12:35:04.102Z”,
data: { scn: 2, reset: false, … } }

{ type: ”click", ts: ISODate(“2015-03-03T12:35:05.312Z”,
data: { x: 23, y: 32, adId: “BB512” } }

{ type: ”close", ts: ISODate(“2015-03-03T12:35:08.774Z”,
data: { snc: 2, logout: true, mostRecent: [ … ] } }

{ type: ”click", ts: ISODate(“2015-03-03T12:35:10.114Z”,
data: { x: 881, y: 913, adId: “F430” } }
```

Question:

What are all the books about databases?

Categories as an Array

```
> db.books.find({ _id : “123” })
{
    _id: “123”,
    title: "MongoDB: The Definitive Guide",
    categories: [“MongoDB”, “Databases”, “Programming”]
}

> db.book.createIndex({categories:1});

> db.books.find({ categories: “Databases” })
```

Question:

How do you apply governance?

Document Validation And Soft Schemas

```
> db.createCollection("books", { "validator":
{ $or: [
  { $and: [ { "v": 1},
            { "title": {$type: “string”} }
             ]
             },
  { $and: [ { "v": 2},
            { "title": {$type: “string”} },
            { "publishDate": {$type: “date”} },
            { $or: [
                    { "thumbnail": {$exists: False}},
                    { "thumbnail": {$type: “binary”}}
                    ]
                    }
             ]
  }
]
});
```

Now THIS is cool…

Summary

- Physical design is different in MongoDB
  - But basic *data* design principles stay the same
- Focus on how an application accesses/manipulates data
- Seek out and capture belongs-to 1:1 relationships
- Use substructure to better align to code objects
- Be polymorphic!
- Evolve the schema to meet requirements as they change

The End

```
{
        "timeFor" :
                {
                        "questions" : "answers"
                },
        "maybe" : "justWrapItUpAlready",
        "definitely" : "lunchTime"
}
```

http://mongodb.com
http://university.mongodb.com