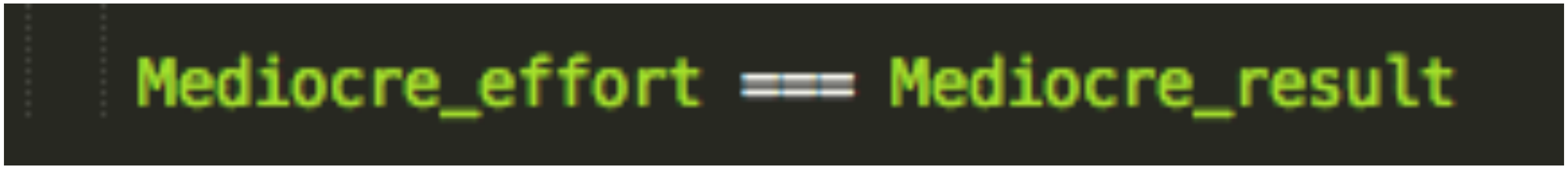
wordnik

Data Modeling for NoSQL

Tony Tam
@fehguy

Data Modeling?



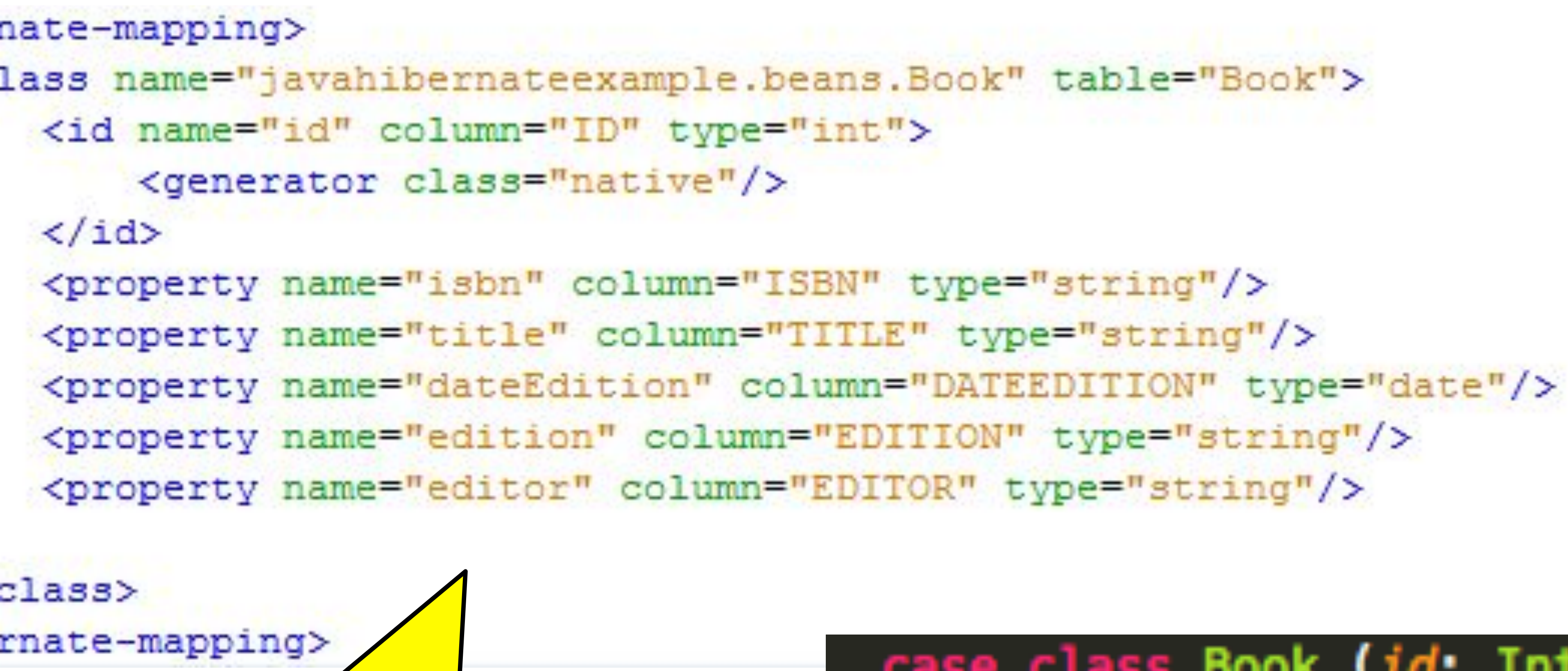
```
Mediocre_effort === Mediocre_result
```

Smart Modeling makes NoSQL work

Why Modeling Matters

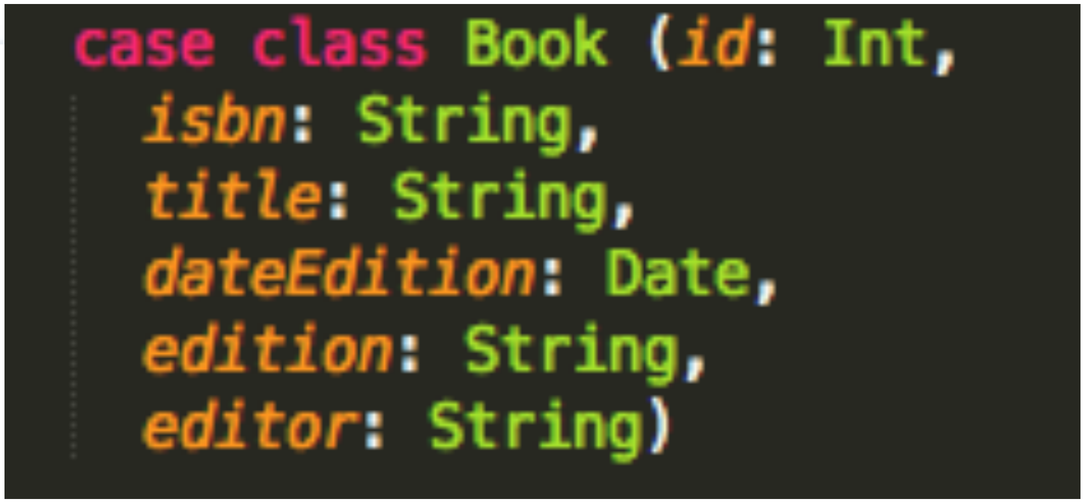
- NoSQL => no joins
- What replaces joins?
  - Hierarchy
  - Duplication of data
  - Different models for querying, indexing
- Your optimal data model is (probably) very different than with relational
  - Simpler
  - More like you develop

Stop Thinking Like This!



```
nate-mapping>
lass name="javahibernateexample.beans.Book" table="Book">
  <id name="id" column="ID" type="int">
      <generator class="native"/>
  </id>
  <property name="isbn" column="ISBN" type="string"/>
  <property name="title" column="TITLE" type="string"/>
  <property name="dateEdition" column="DATEEDITION" type="date"/>
  <property name="edition" column="EDITION" type="string"/>
  <property name="editor" column="EDITOR" type="string"/>

class>
rnate-mapping>
```

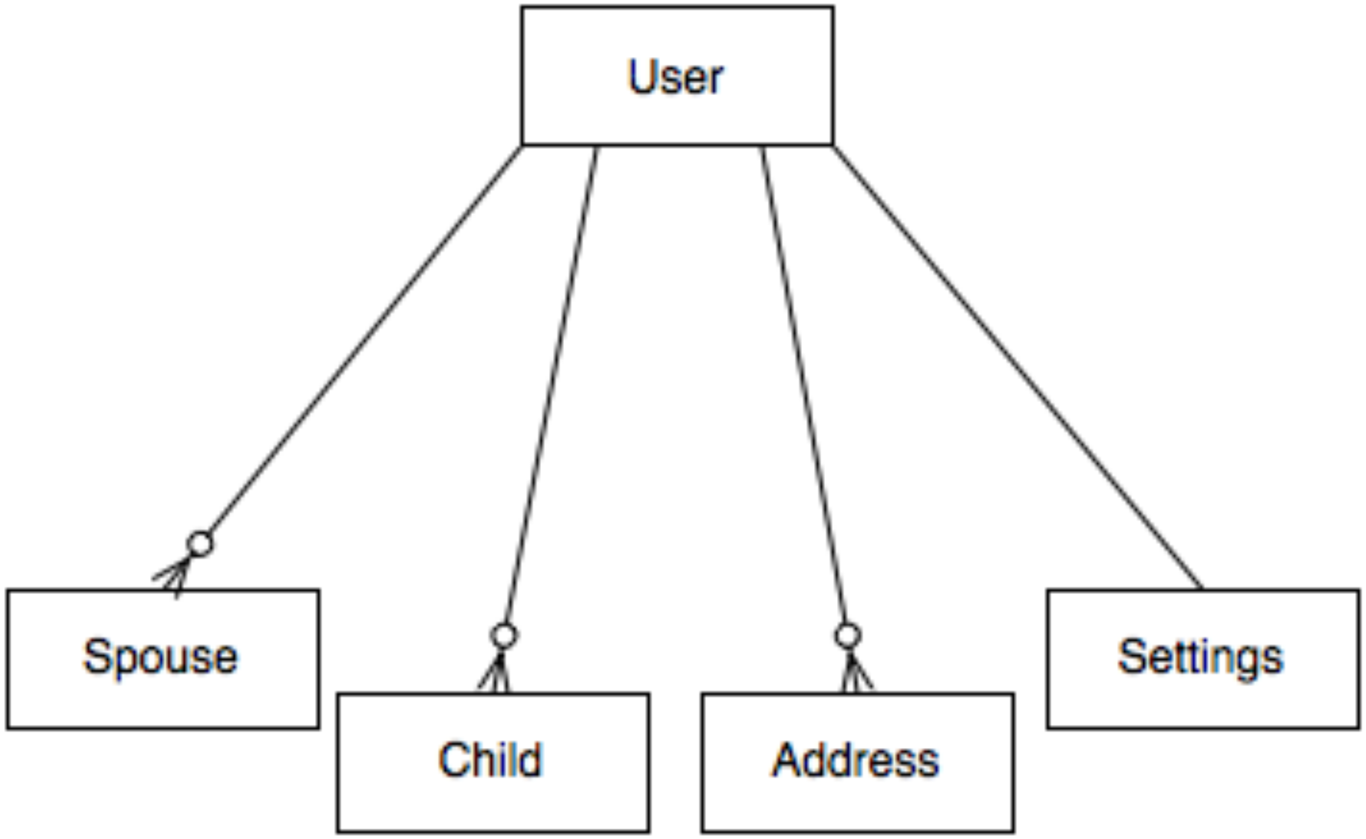
endless layers
of abstraction
(and misery)



```
case class Book (id: Int,
   isbn: String,
   title: String,
   dateEdition: Date,
   edition: String,
   editor: String)
```

Hierarchy before NoSQL

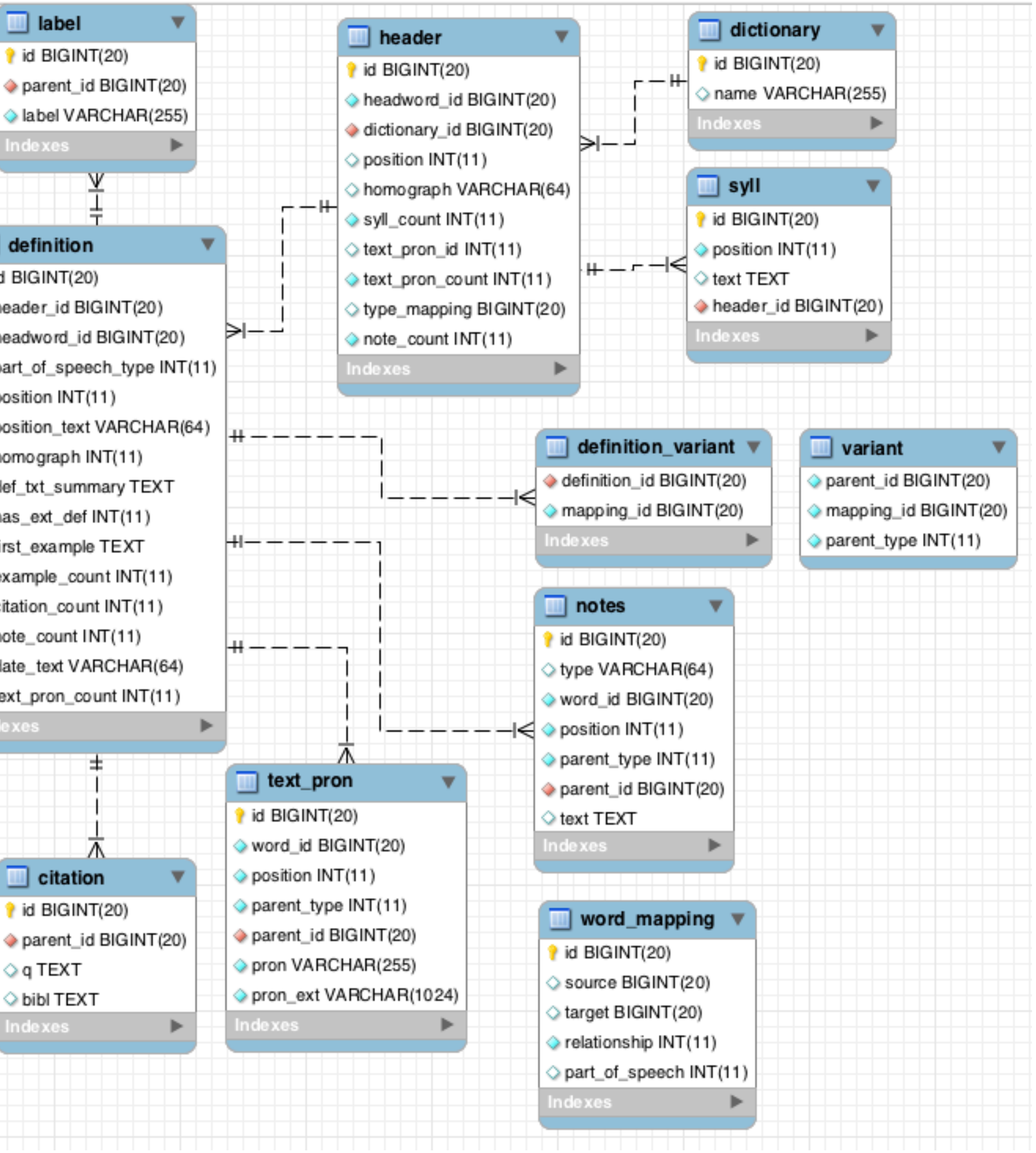
- Simple User Model

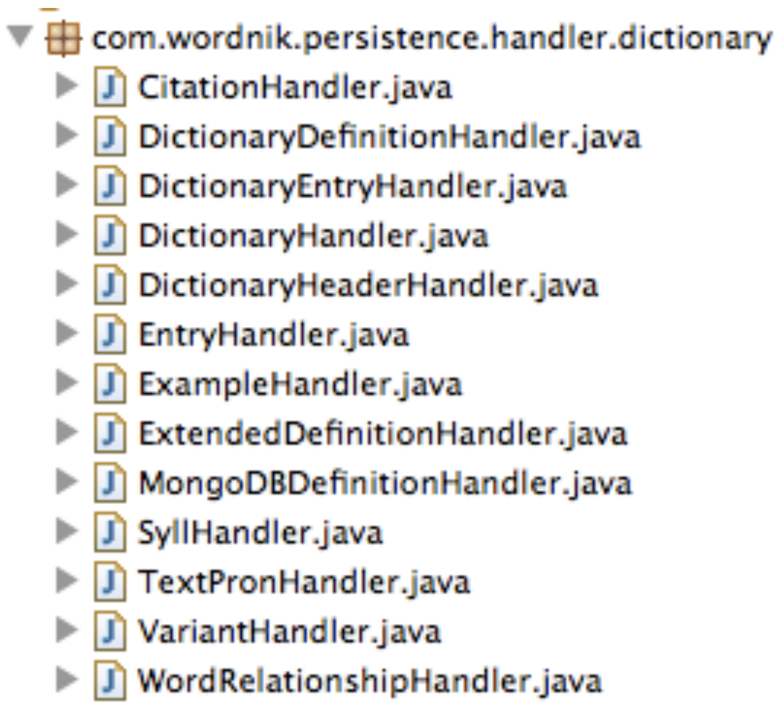


Hierarchy before NoSQL

- Tuned Queries
  - Write some brittle SQL:
    - “select user.id, … inner join settings on …
  - Pick out the fields and construct object hierarchy (this gets nasty, fast)
    - (outer joins for optional values?)
- Object fetching
  - Queries follow object graph, PK/FK
  - 5 queries to fetch object in this example

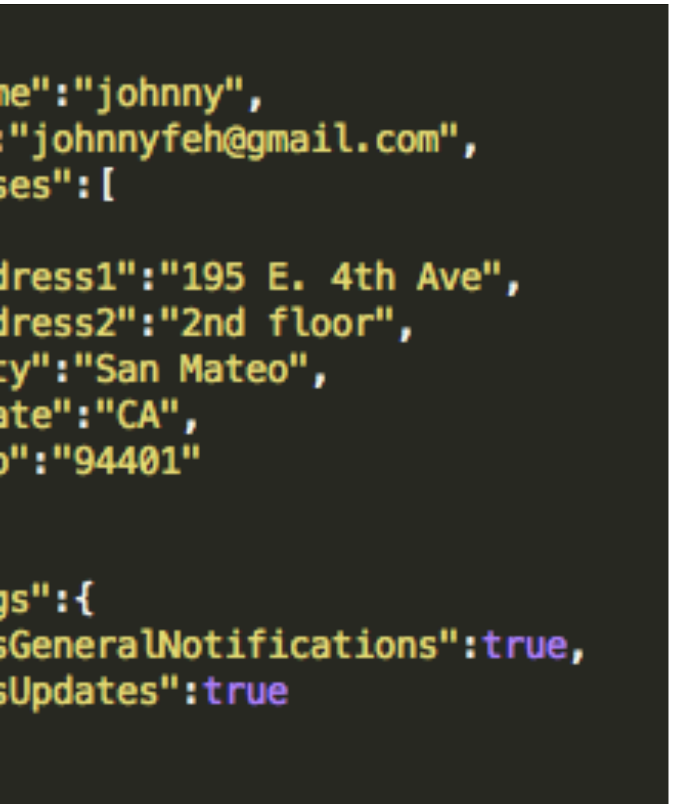
Hierarchy before NoSQL



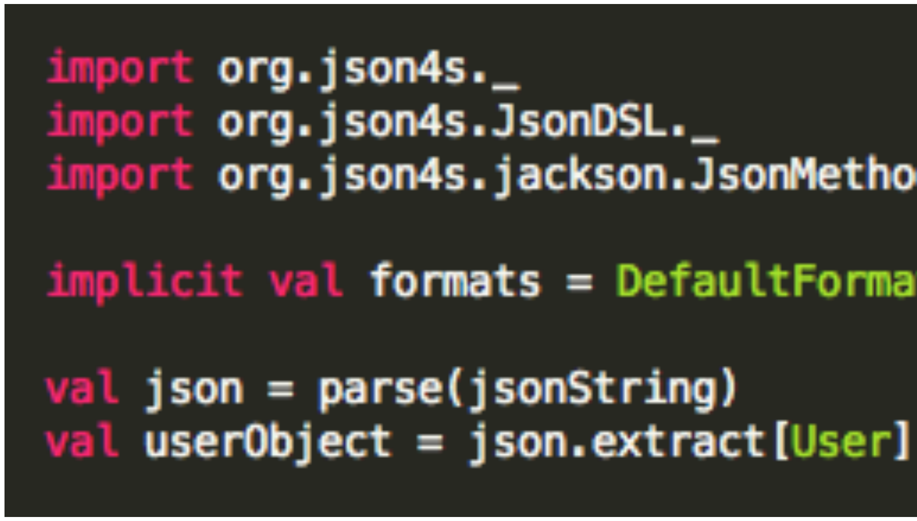

- com.wordnik.persistence.handler.dictionary
  - CitationHandler.java
  - DictionaryDefinitionHandler.java
  - DictionaryEntryHandler.java
  - DictionaryHandler.java
  - DictionaryHeaderHandler.java
  - EntryHandler.java
  - ExampleHandler.java
  - ExtendedDefinitionHandler.java
  - MongoDBDefinitionHandler.java
  - SyllHandler.java
  - TextPronHandler.java
  - VariantHandler.java
  - WordRelationshipHandler.java

Hierarchy with NoSQL

- JSON structure mapped to objects
  - Fetch json from MongoDB**
  - Unmarshall into objects/tuples
  - Use it

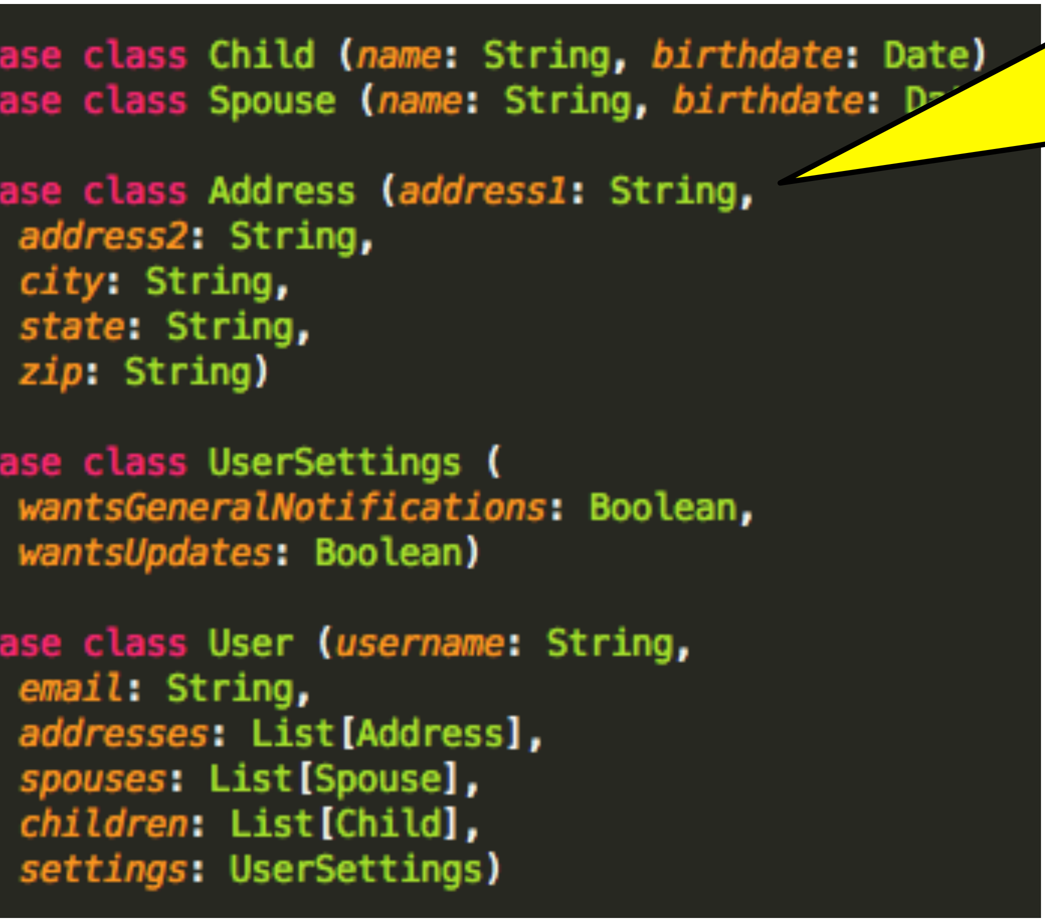


```
me":"johnny",
:"johnnyfeh@gmail.com",
ses":[

dress1":"195 E. 4th Ave",
dress2":"2nd floor",
ty":"San Mateo",
ate":"CA",
p":"94401"


gs":{
sGeneralNotifications":true,
sUpdates":true
```



```
import org.json4s._
import org.json4s.JsonDSL._
import org.json4s.jackson.JsonMetho

implicit val formats = DefaultForma

val json = parse(jsonString)
val userObject = json.extract[User]
```

Hierarchy with NoSQL

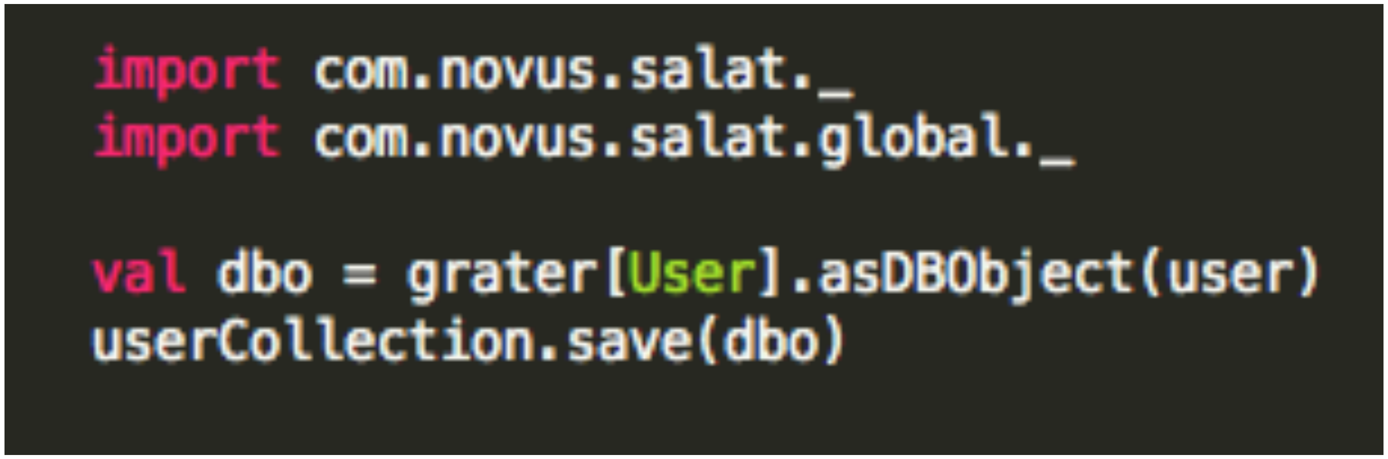


```
ase class Child (name: String, birthdate: Date)
ase class Spouse (name: String, birthdate: Da

ase class Address (address1: String,
 address2: String,
 city: String,
 state: String,
 zip: String)

ase class UserSettings (
 wantsGeneralNotifications: Boolean,
 wantsUpdates: Boolean)

ase class User (username: String,
 email: String,
 addresses: List[Address],
 spouses: List[Spouse],
 children: List[Child],
 settings: UserSettings)
```

Focus on your Software, not DB layer!

Hierarchy with NoSQL

- Write operations
  - Atomic upsert (create, update or fail)

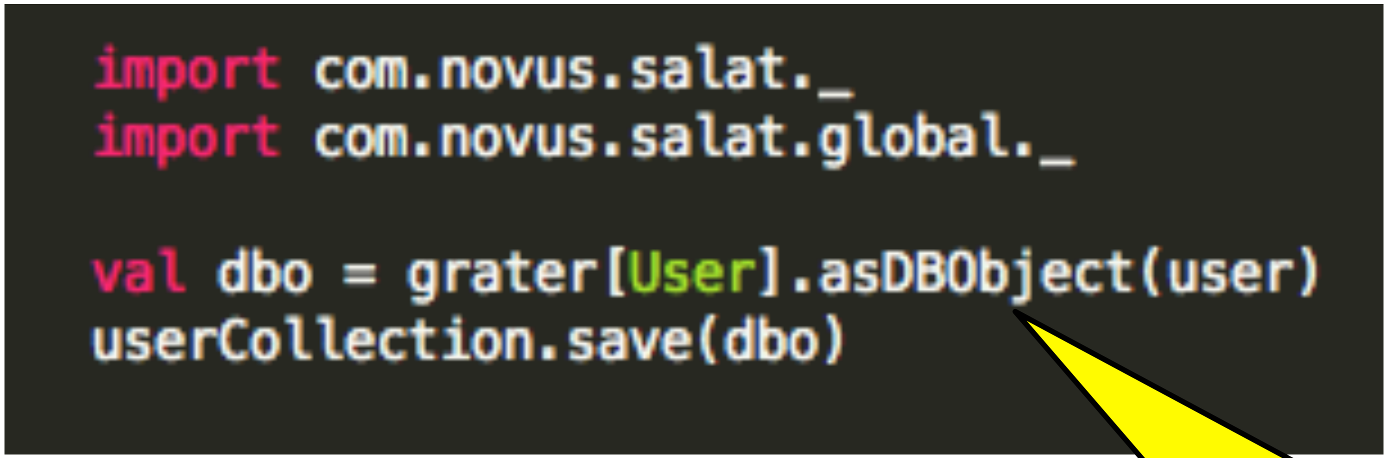


```
import com.novus.salat._
import com.novus.salat.global._

val dbo = grater[User].asDBObject(user)
userCollection.save(dbo)
```

  - Saves all levels of object atomically
  - *Reduces* need for transactions

Hierarchy with NoSQL

- Write operations
  - Atomic upsert (create, update or fail)



```
import com.novus.salat._
import com.novus.salat.global._

val dbo = grater[User].asDBObject(user)
userCollection.save(dbo)
```

  - Saves all levels of object atomically
  - *Reduces* need for transactions

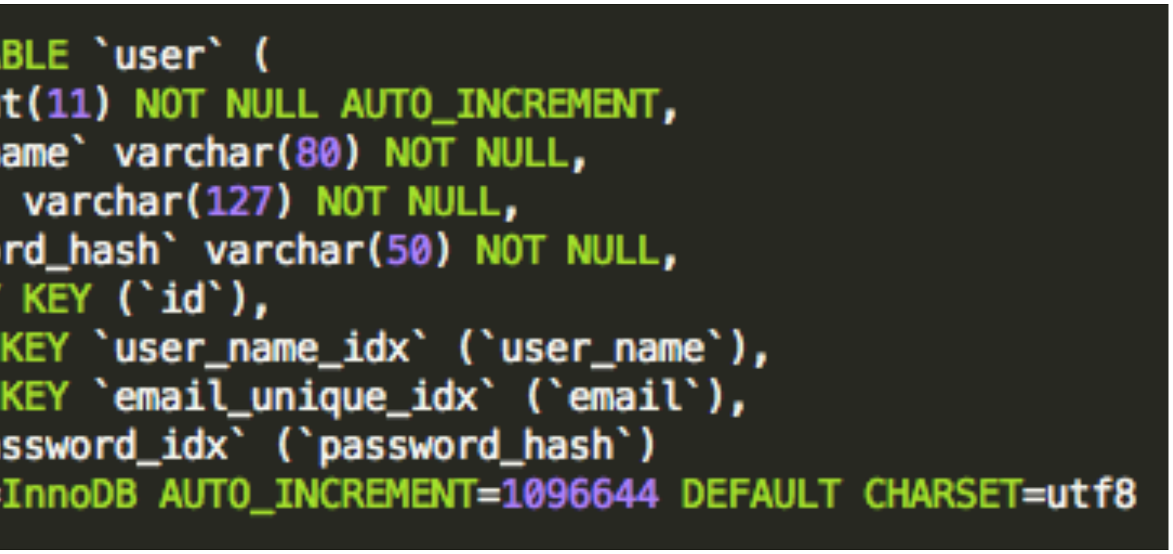
Convenienc
not magic

All or
othing

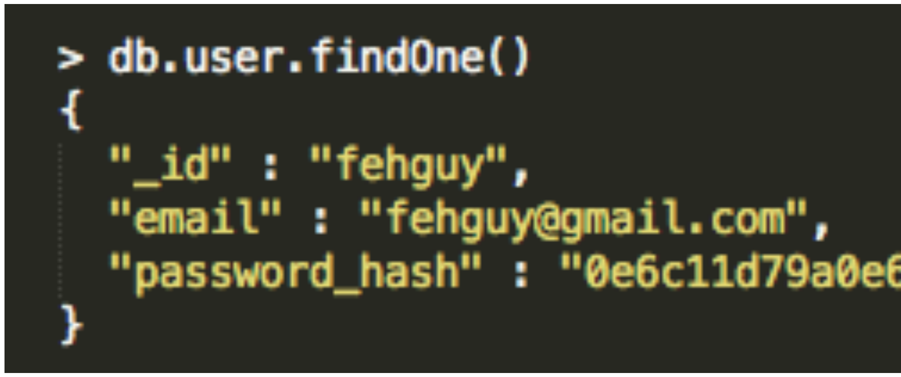
Unique Identifiers in your Data

- Relational design => PK/FK
  - Often not “meaningful” identifiers for data

- User Data Model



```
BLE `user` (
t(11) NOT NULL AUTO_INCREMENT,
ame` varchar(80) NOT NULL,
 varchar(127) NOT NULL,
rd_hash` varchar(50) NOT NULL,
 KEY (`id`),
KEY `user_name_idx` (`user_name`),
KEY `email_unique_idx` (`email`),
ssword_idx` (`password_hash`)
InnoDB AUTO_INCREMENT=1096644 DEFAULT CHARSET=utf8
```

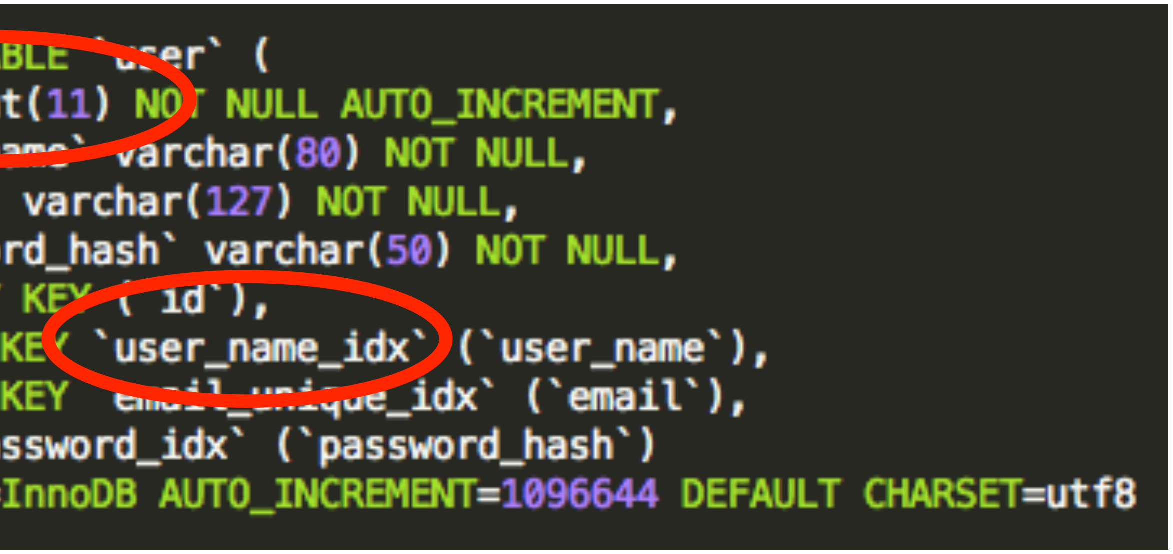


```
> db.user.findOne()
{
    "_id" : "fehguy",
    "email" : "fehguy@gmail.com",
    "password_hash" : "0e6c11d79a0e6
}
```

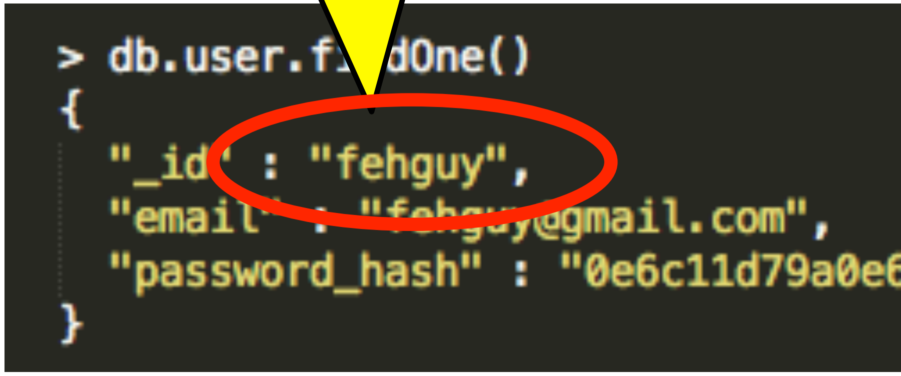
Unique Identifiers in your Data

- Relational design => PK/FK
  - Often not “meaningful” identifiers for data
- User Data Model

Unique by username



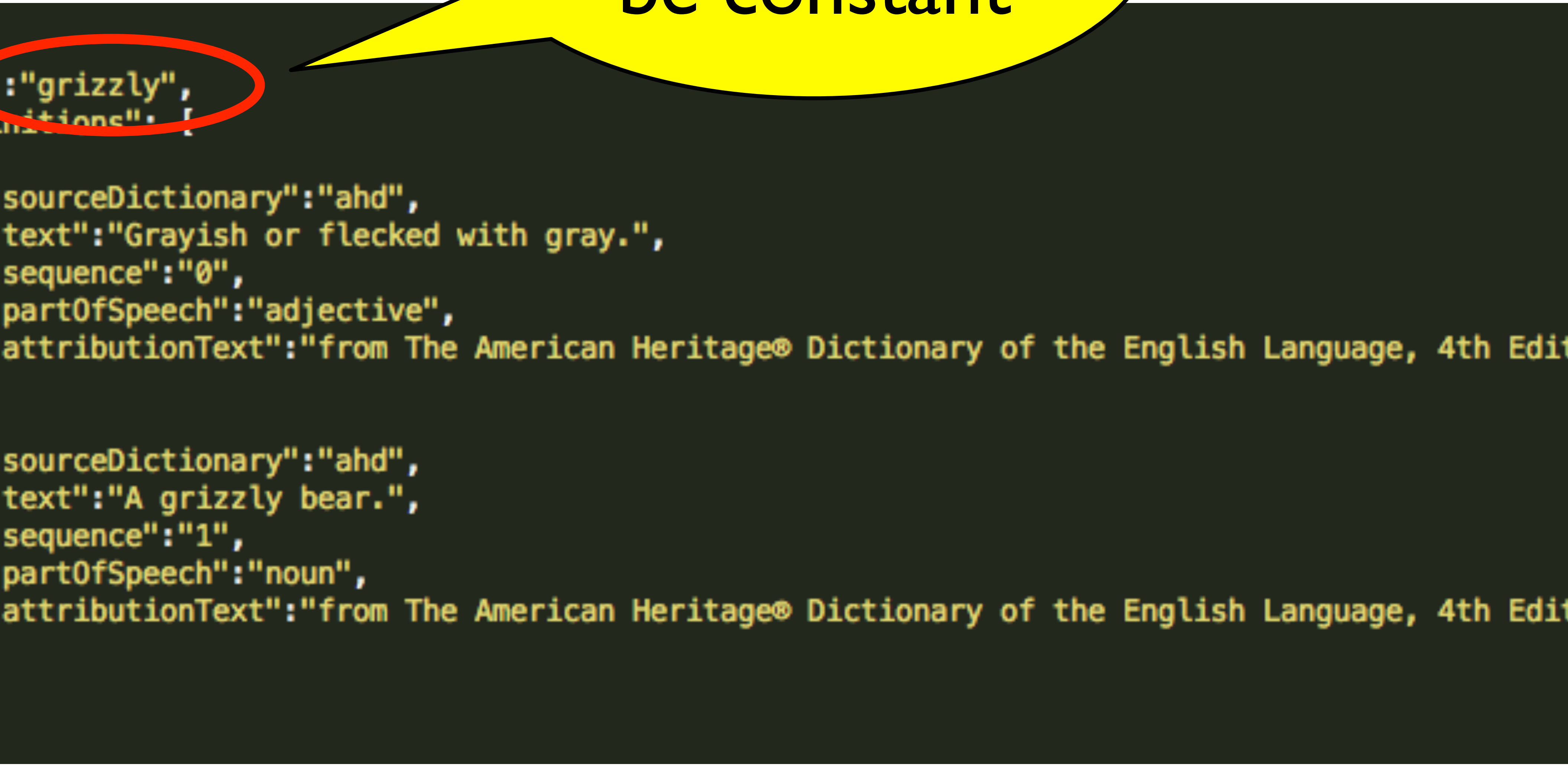
```
BLE `user` (
t(11) NOT NULL AUTO_INCREMENT,
ame` varchar(80) NOT NULL,
 varchar(127) NOT NULL,
rd_hash` varchar(50) NOT NULL,
 KEY (`id`),
KEY `user_name_idx` (`user_name`),
KEY `email_unique_idx` (`email`),
ssword_idx` (`password_hash`)
InnoDB AUTO_INCREMENT=1096644 DEFAULT CHARSET=utf8
```



```
> db.user.findOne()
{
	"_id" : "fehguy",
	"email" : "fehguy@gmail.com",
	"password_hash" : "0e6c11d79a0e6
}
```

Unique Identifiers in your Data

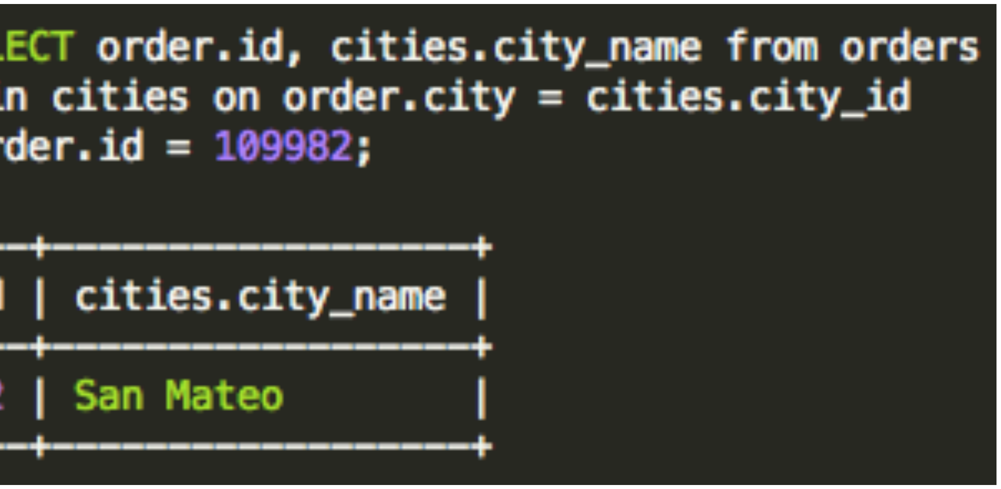
- Words



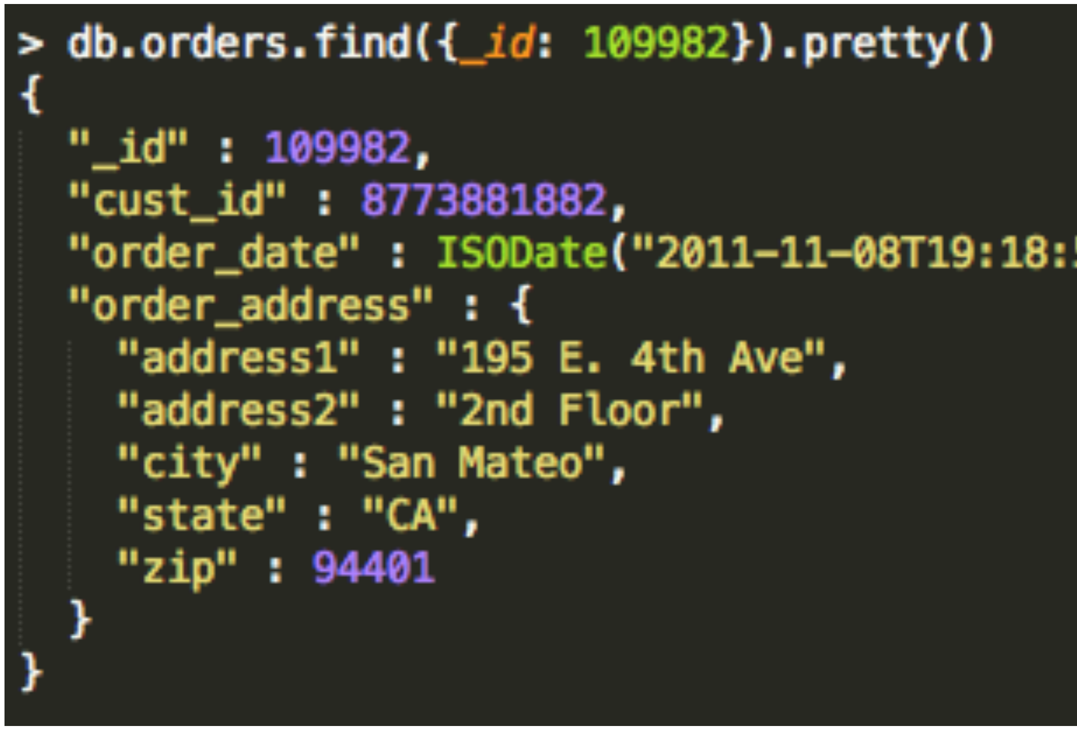
```
:"grizzly",
itions": [

sourceDictionary":"ahd",
text":"Grayish or flecked with gray.",
sequence":"0",
partOfSpeech":"adjective",
attributionText":"from The American Heritage® Dictionary of the English Language, 4th Edit


sourceDictionary":"ahd",
text":"A grizzly bear.",
sequence":"1",
partOfSpeech":"noun",
attributionText":"from The American Heritage® Dictionary of the English Language, 4th Edit
```

Data Duplication

- Without Joins, what about SQL lookup tables?
  - Duplication of data in NoSQL is *required*
- Trade storage for speed



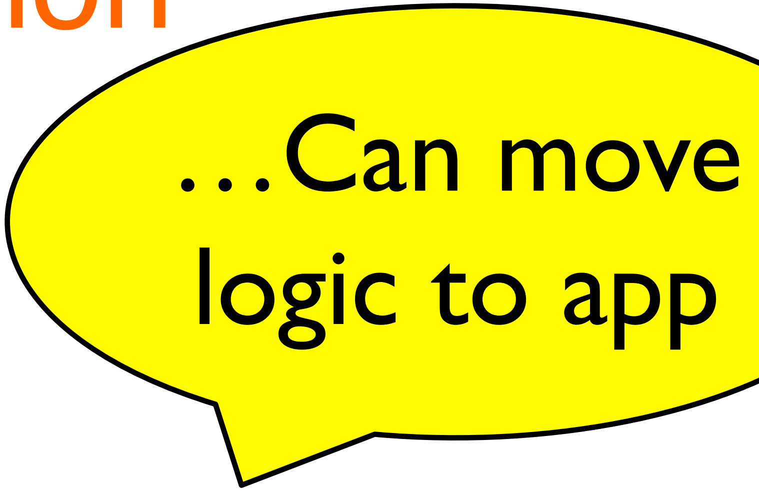
```
LECT order.id, cities.city_name from orders
.n cities on order.city = cities.city_id
rder.id = 109982;

--+--------------------+
l | cities.city_name |
--+--------------------+
2 | San Mateo          |
--+--------------------+
```

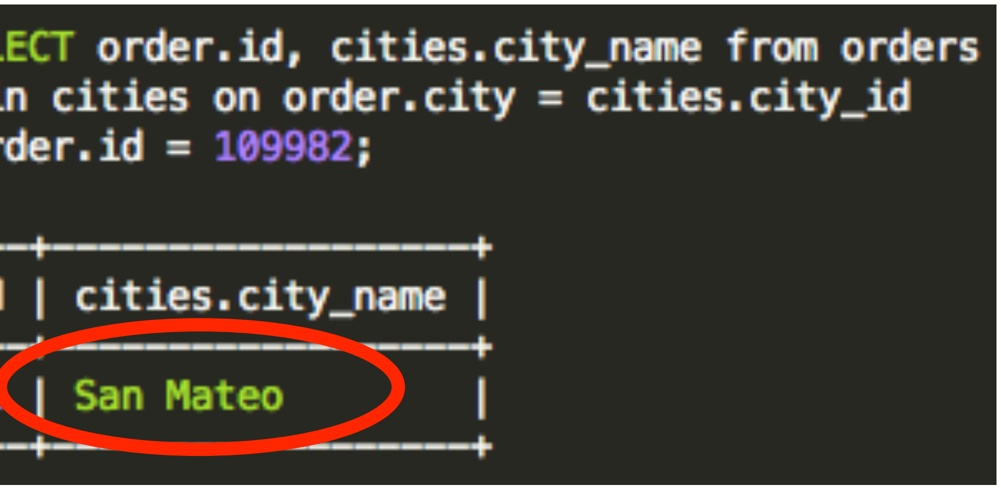


```
> db.orders.find({_id: 109982}).pretty()
{
  "_id" : 109982,
  "cust_id" : 8773881882,
  "order_date" : ISODate("2011-11-08T19:18:5
  "order_address" : {
    "address1" : "195 E. 4th Ave",
    "address2" : "2nd Floor",
    "city" : "San Mateo",
    "state" : "CA",
    "zip" : 94401
  }
}
```

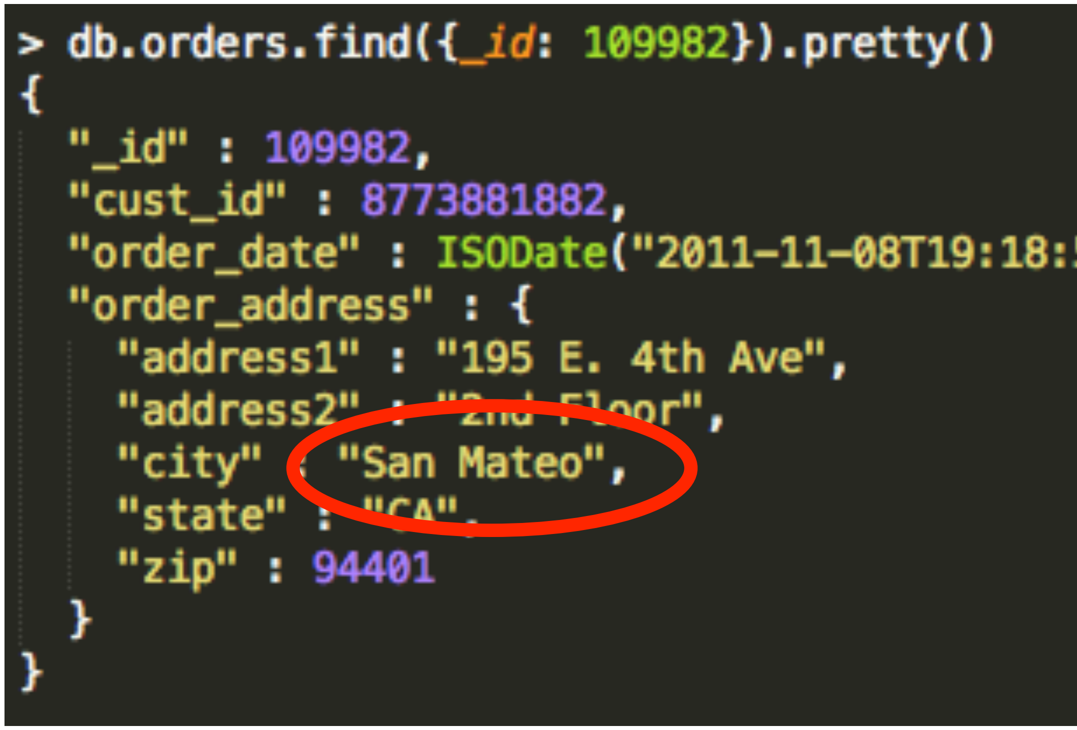
Data Duplication

- Without Joins, what about tables?
  - Duplication of data in NoSQL is *required*
- Trade storage for speed





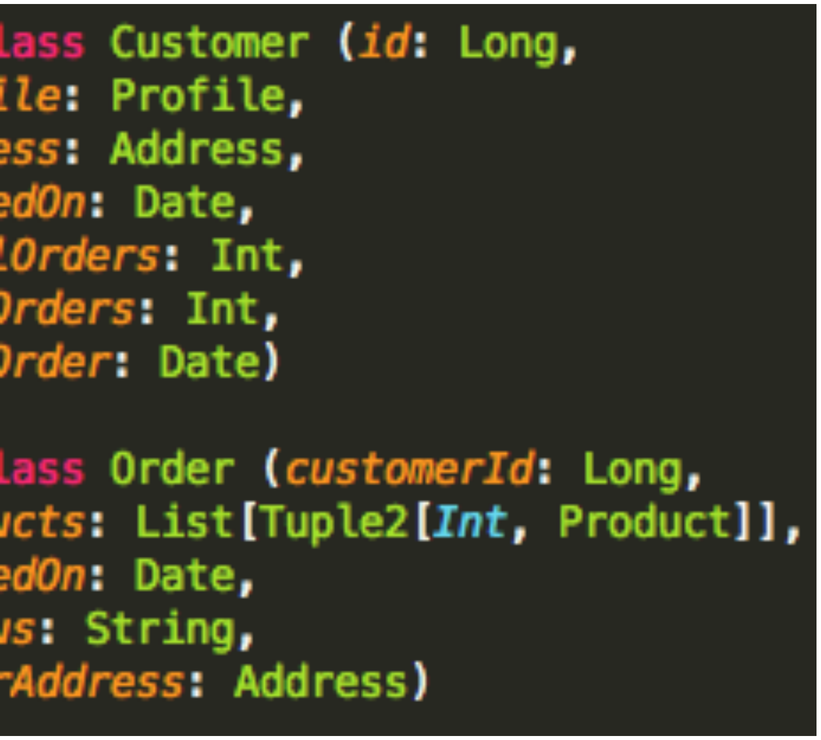
```
LECT order.id, cities.city_name from orders
in cities on order.city = cities.city_id
rder.id = 109982;

--+--------------------+
l | cities.city_name |
--+--------------------+
  | San Mateo          |
--+--------------------+
```



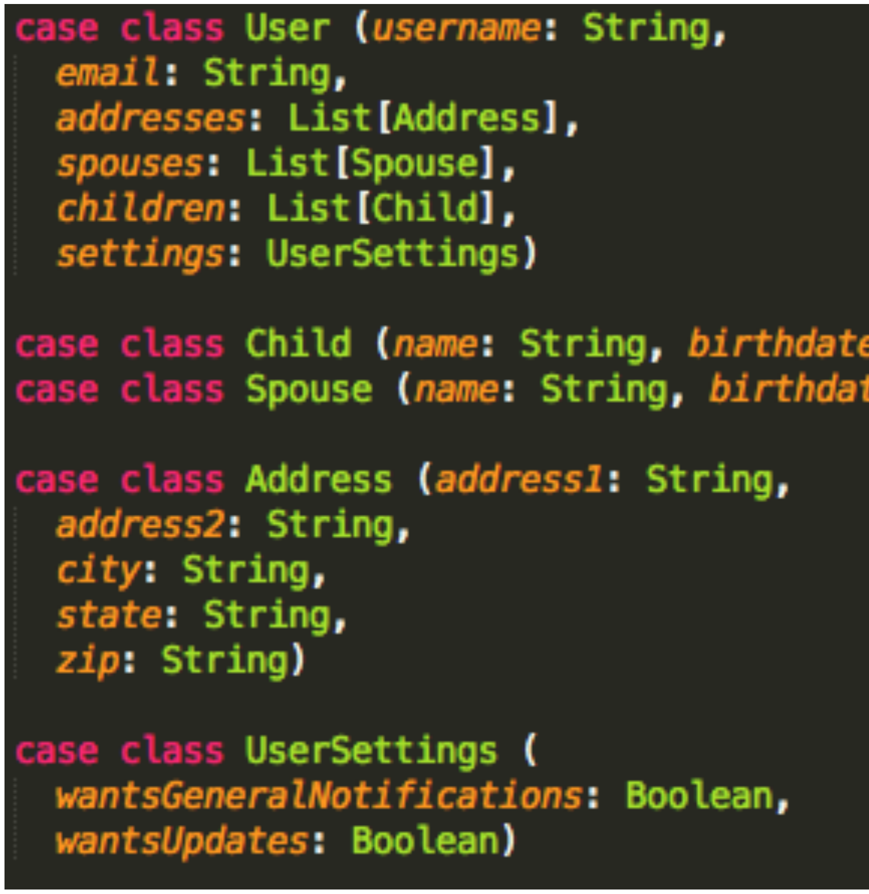
```
> db.orders.find({_id: 109982}).pretty()
{
  "_id" : 109982,
  "cust_id" : 8773881882,
  "order_date" : ISODate("2011-11-08T19:18:5
  "order_address" : {
    "address1" : "195 E. 4th Ave",
    "address2" : "2nd Floor",
    "city" : "San Mateo",
    "state" : "CA",
    "zip" : 94401
  }
}
```

Data Duplication

- Many fields don’t change, *ever*
- But… many do
  - New decisions for the developer!
  - Often background updates



```
lass Customer (id: Long,
ile: Profile,
ess: Address,
edOn: Date,
lOrders: Int,
Orders: Int,
Order: Date)

lass Order (customerId: Long,
ucts: List[Tuple2[Int, Product]],
edOn: Date,
us: String,
rAddress: Address)
```

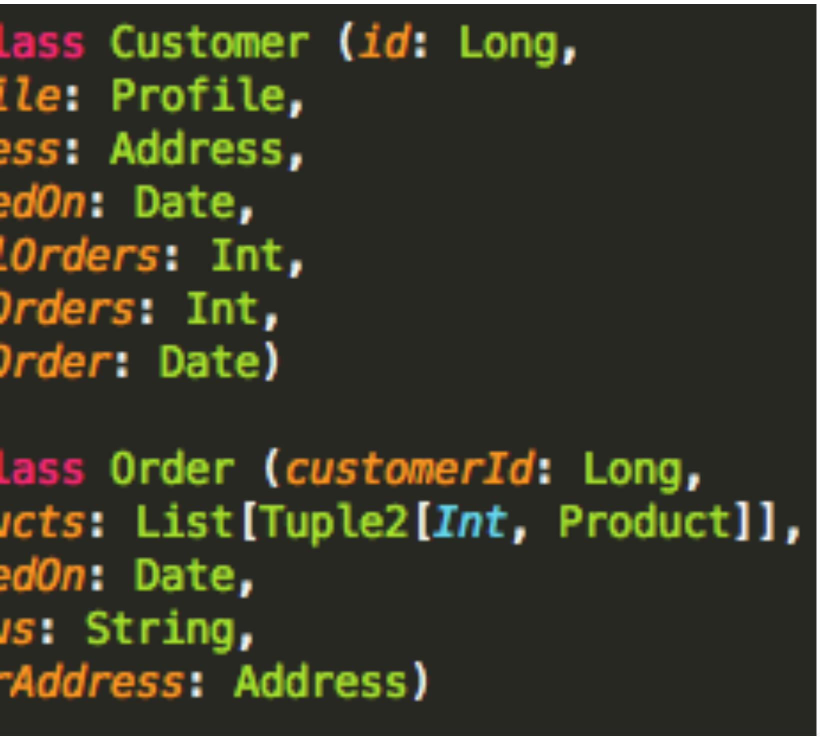


```
case class User (username: String,
  email: String,
  addresses: List[Address],
  spouses: List[Spouse],
  children: List[Child],
  settings: UserSettings)

case class Child (name: String, birthdate
case class Spouse (name: String, birthdat

case class Address (address1: String,
  address2: String,
  city: String,
  state: String,
  zip: String)

case class UserSettings (
  wantsGeneralNotifications: Boolean,
  wantsUpdates: Boolean)
```

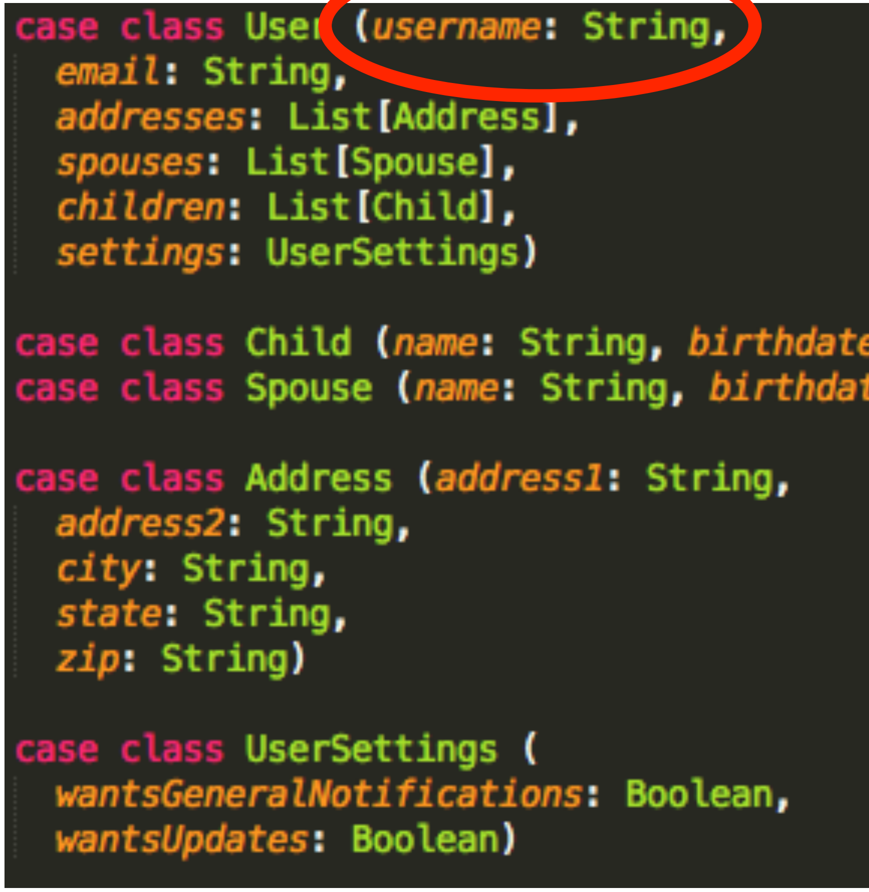
Data Duplication

- Many fields don't change,
- But… many do
  - New decisions for the developer!
  - Often background updates

How often does this change?

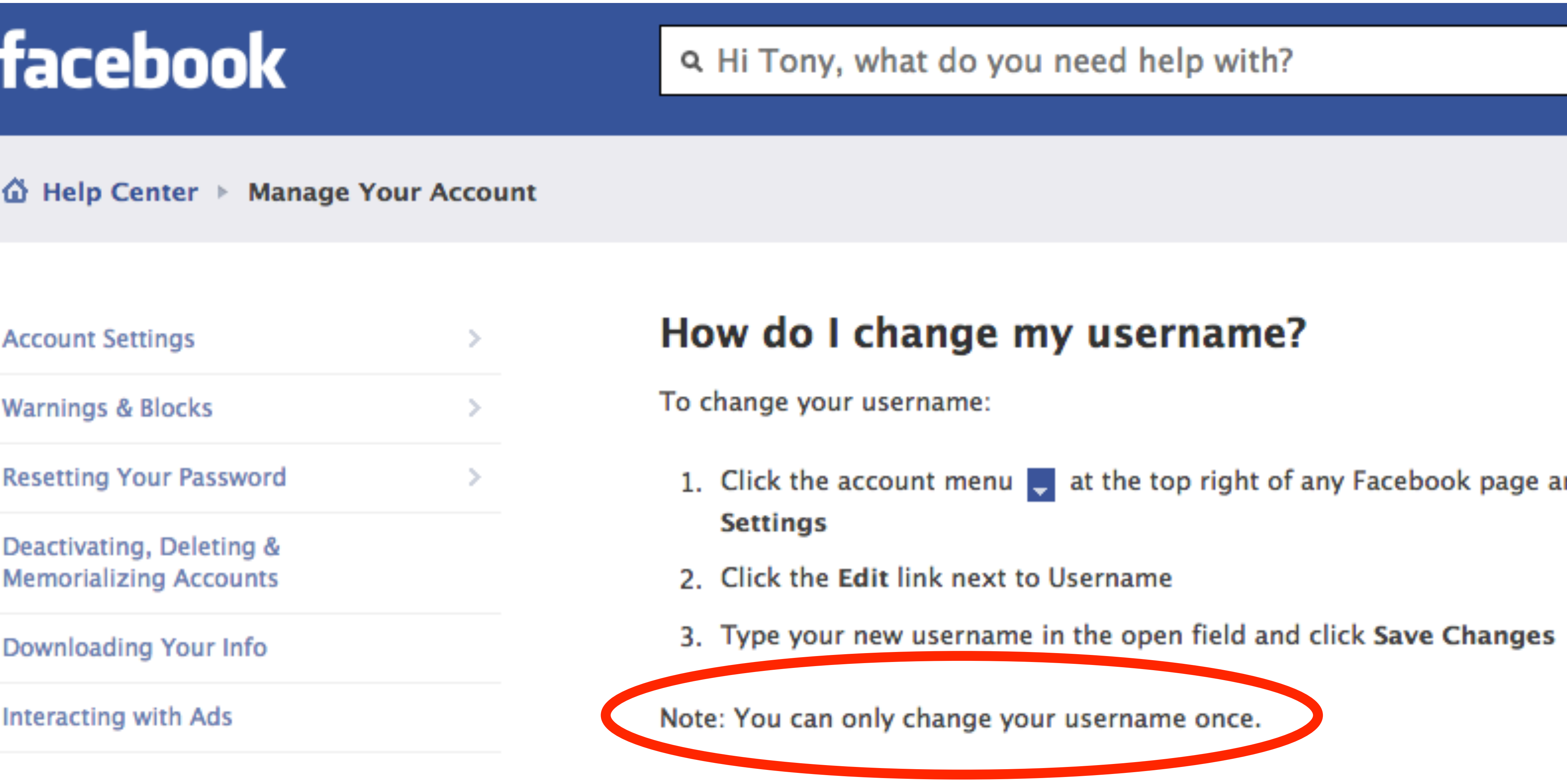


```
lass Customer (id: Long,
ile: Profile,
ess: Address,
edOn: Date,
lOrders: Int,
Orders: Int,
Order: Date)

lass Order (customerId: Long,
ucts: List[Tuple2[Int, Product]],
edOn: Date,
us: String,
rAddress: Address)
```



```
case class User (username: String,
  email: String,
  addresses: List[Address],
  spouses: List[Spouse],
  children: List[Child],
  settings: UserSettings)

case class Child (name: String, birthdate
case class Spouse (name: String, birthdat

case class Address (address1: String,
  address2: String,
  city: String,
  state: String,
  zip: String)

case class UserSettings (
  wantsGeneralNotifications: Boolean,
  wantsUpdates: Boolean)
```

Data Duplication



facebook

Hi Tony, what do you need help with?

Help Center ▸ Manage Your Account

How do I change my username?

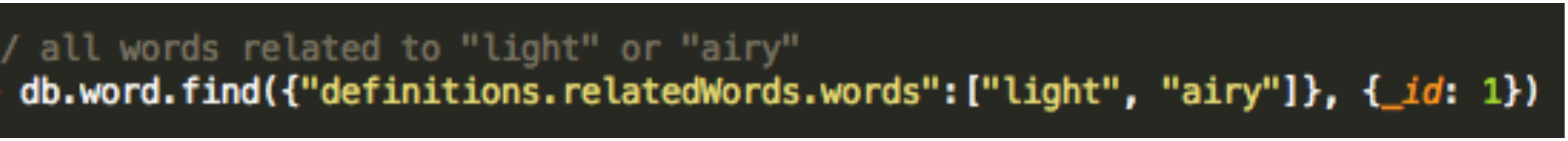
To change your username:

1. Click the account menu at the top right of any Facebook page a **Settings**
2. Click the **Edit** link next to Username
3. Type your new username in the open field and click **Save Changes**

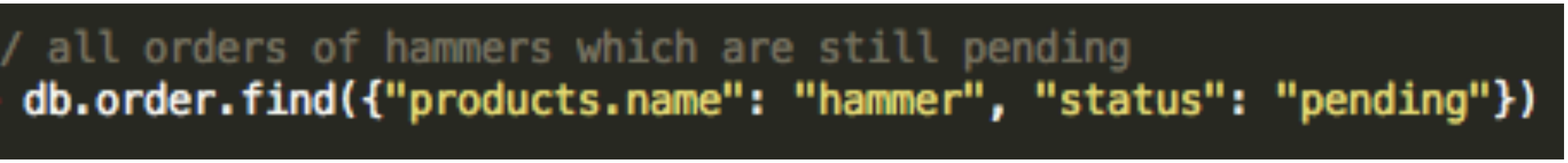
Note: You can only change your username once.

Reaching into Objects

- Incredible feature of MongoDB
  - Dot syntax *safely*** traverses the object graph



```
/ all words related to "light" or "airy"
 db.word.find({"definitions.relatedWords.words":["light", "airy"]}, {_id: 1})
```

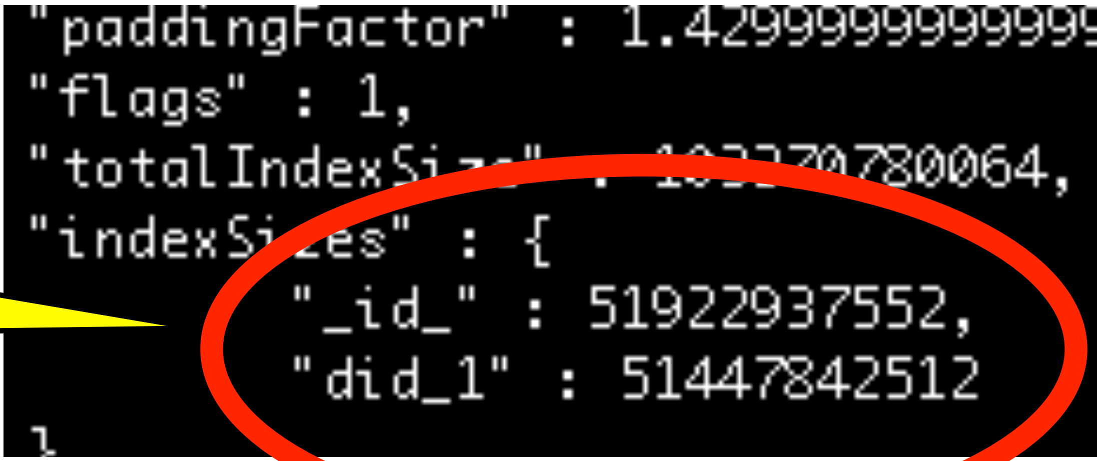


```
/ all orders of hammers which are still pending
 db.order.find({"products.name": "hammer", "status": "pending"})
```

Inner Indexes

- Convenience at a cost
  - No index => table scan
  - No value? => table scan
  - No child value? => table scan
- Table scan with big collection?
- Can't index everything!

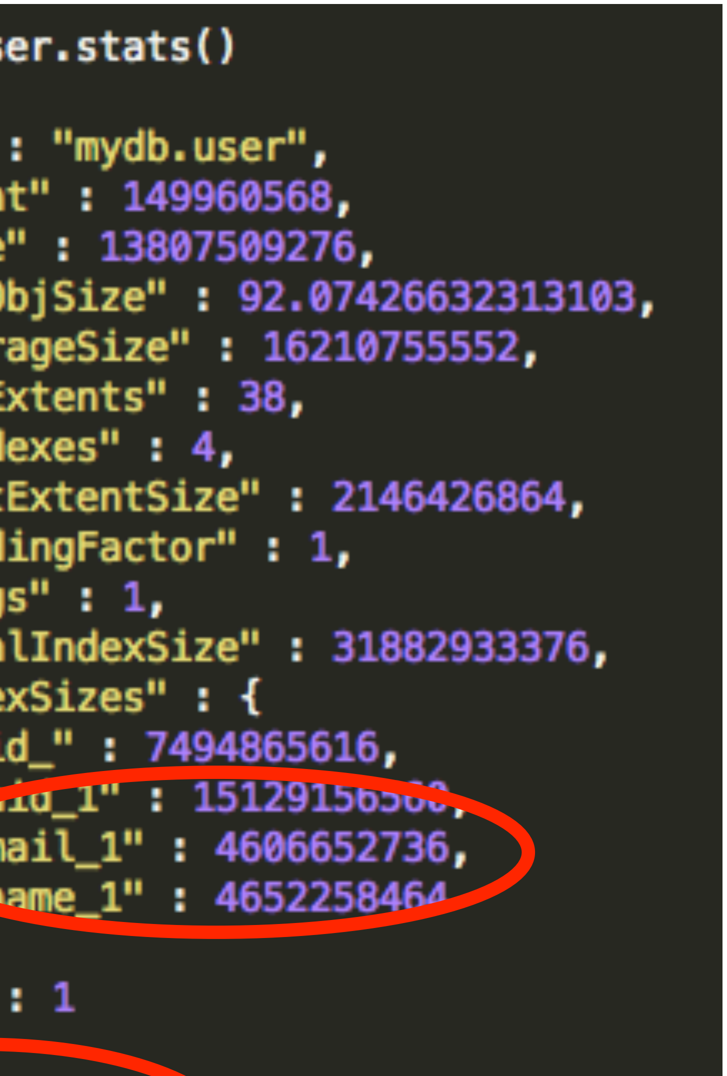
96GB of Indexes?



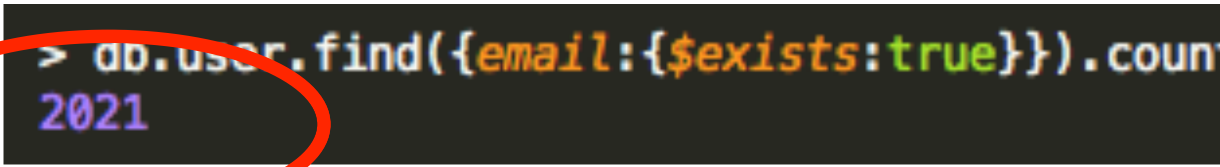
```
"paddingFactor" : 1.4299999999999
"flags" : 1,
"totalIndexSize" : 103370780064,
"indexSizes" : {
        "_id_" : 51922937552,
        "did_1" : 51447842512
}
```

Inner Indexes

- This ~~will~~ *should* drive your Data Model
- Sparse Data test



```
er.stats()

: "mydb.user",
t" : 149960568,
e" : 13807509276,
ObjSize" : 92.07426632313103,
rageSize" : 16210755552,
Extents" : 38,
lexes" : 4,
ExtentSize" : 2146426864,
dingFactor" : 1,
gs" : 1,
alIndexSize" : 31882933376,
exSizes" : {
id_" : 7494865616,
id_1" : 15129156560,
ail_1" : 4606652736,
ame_1" : 4652258464

: 1
```



```
> db.user.find({email:{$exists:true}}).coun
2021
```

Even with only 2000 non-empty values!

Adding & Modifying

- Append in mongo is blazing fast
  - “tail” of data is *always* in memory
  - Pre-allocated data files
- Main expense is “index maintenance”
  - Some marshalling/unmarshalling cost**
- Modifying? Object growth
  - Pre-allocation of space built in collection design

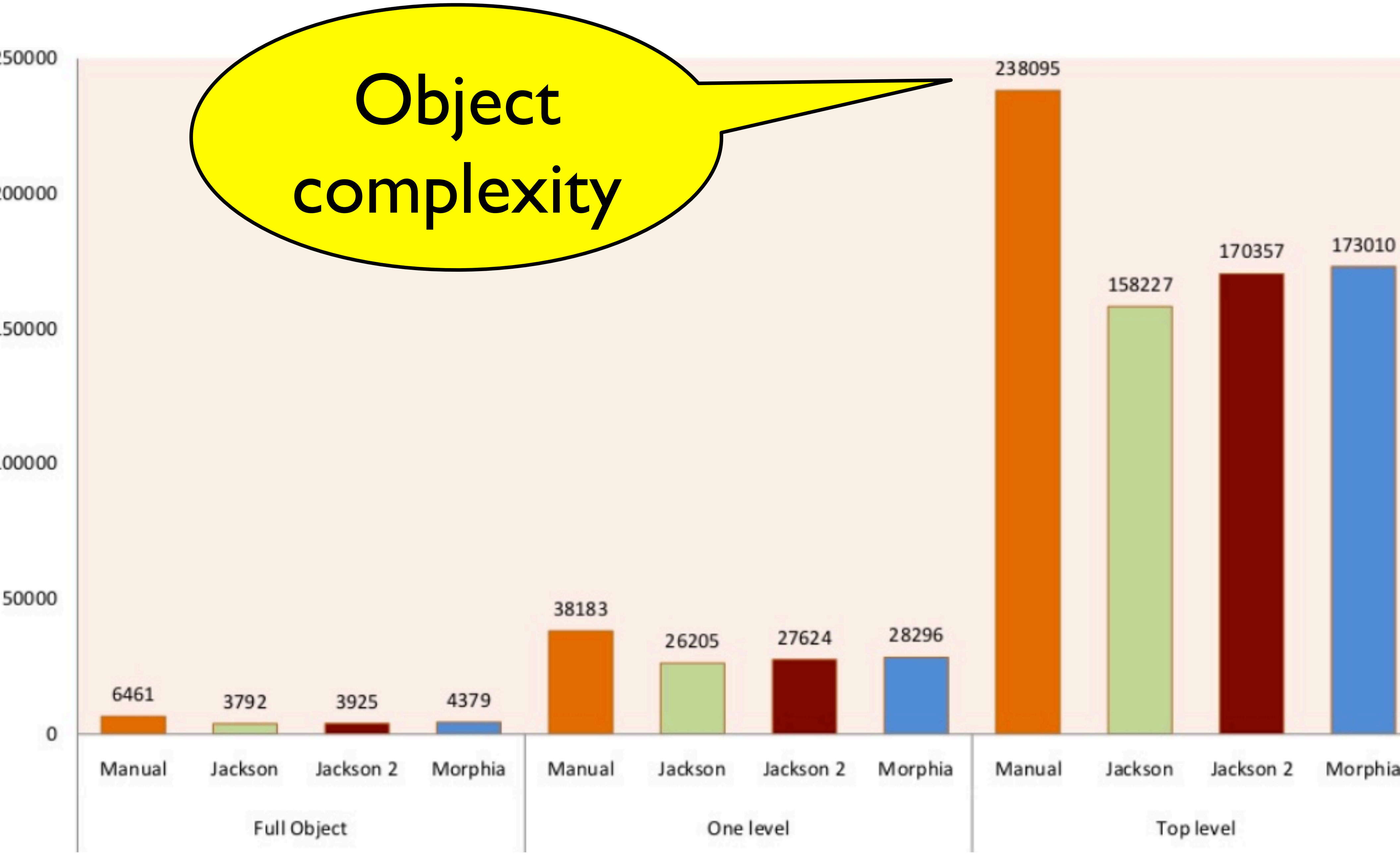
Adding & Modifying

- Each object has allocated space
  - Exceed that space, need to relocate object
  - Leaves “hole” in collection
- Large increases to documents hurts your overall performance
- Your data model should strive for equally-sized objects as much as possible

Retrieval

- Many same rules apply as relational
- Indexes
  - complex/inner or not
  - Indexes in RAM? Yes
  - Cardinality matters
- New(ish) considerations
  - Complex hierarchy not free
    - Marshalling ⇔ unmarshalling

Marshalling & Unmarshalling



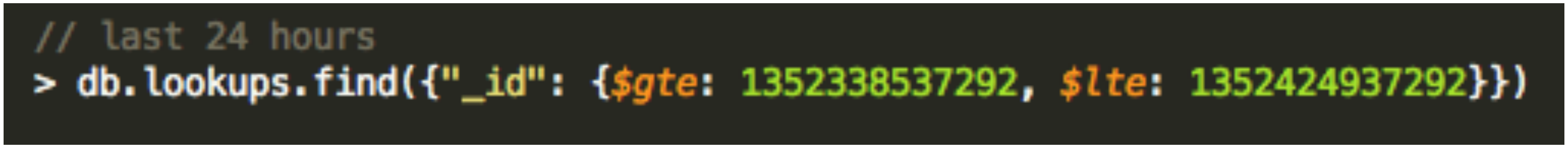
Marshalling & Unmarshalling

- All you can eat from your Data Model?
- Techniques have tremendous impact
    - Development ease until it matters
    - 50% speed bump with manual mapping

Only demand what you can consume!

Making the most of _id

- Indexes matter
- Tailor your _id to be meaningful by access pattern
  - It’s your first defense when auto-sharding
- Date-driven data?
  - Monotonically _id value

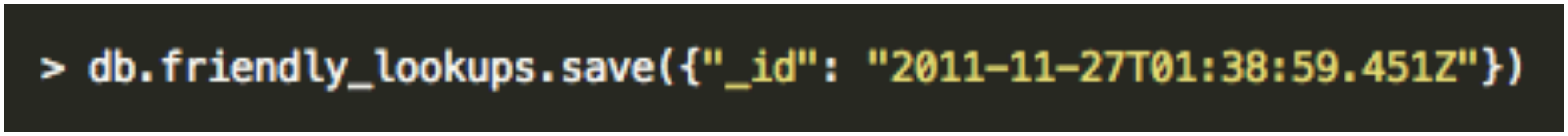


```
// last 24 hours
> db.lookups.find({"_id": {$gte: 1352338537292, $lte: 1352424937292}})
```

  - Ensures recent data is “hot”

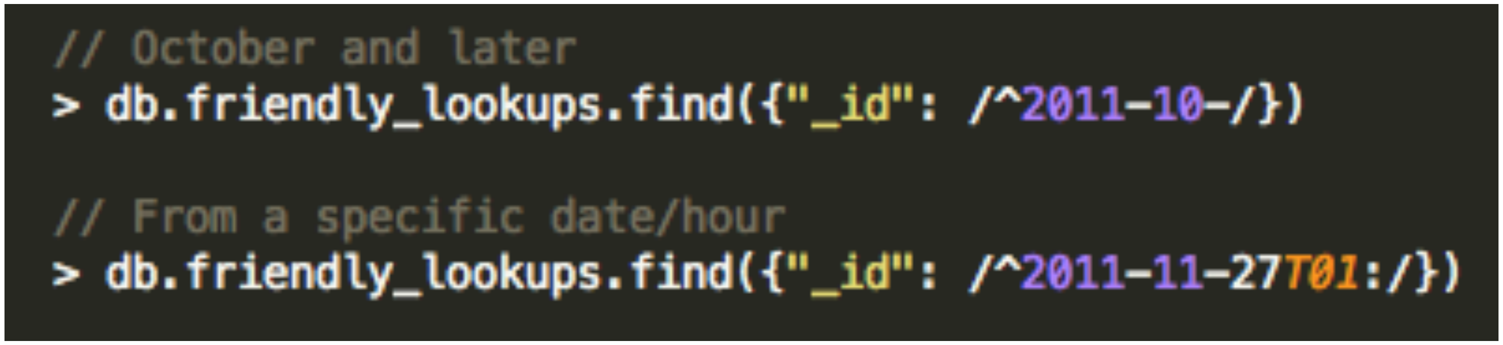
Making the most of _id

- Other time-based data techniques



```
> db.friendly_lookups.save({"_id": "2011-11-27T01:38:59.451Z"})
```

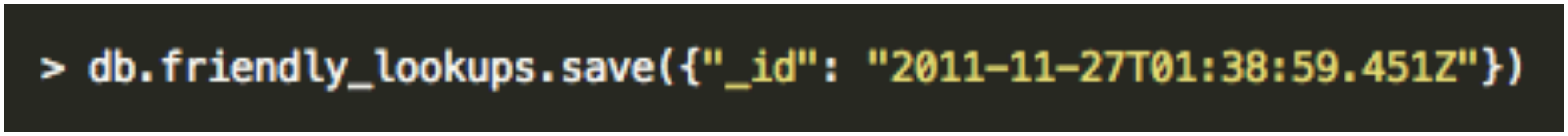
- Flexibility in querying



```
// October and later
> db.friendly_lookups.find({"_id": /^2011-10-/})

// From a specific date/hour
> db.friendly_lookups.find({"_id": /^2011-11-27T01:/})
```

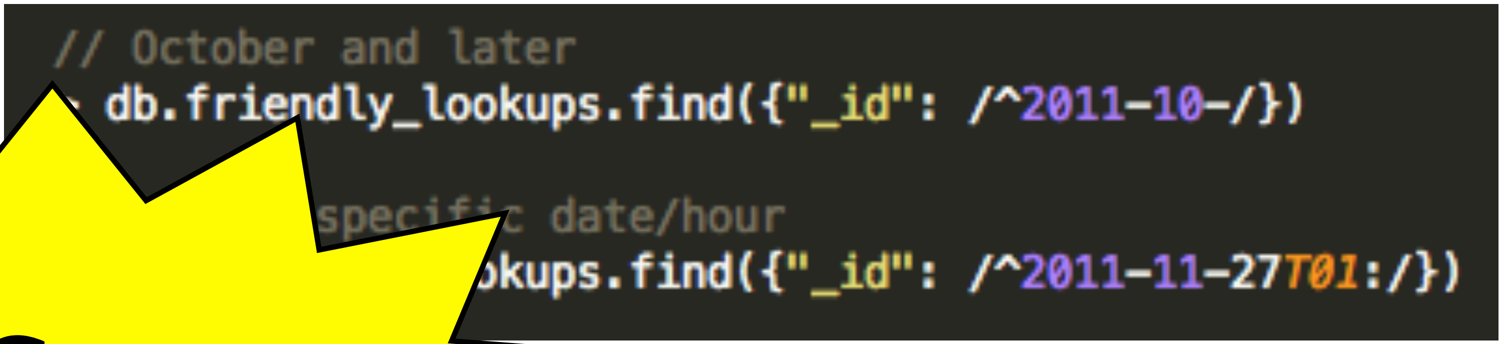
Making the most of _id

- Other time-based data techniques



```
> db.friendly_lookups.save({"_id": "2011-11-27T01:38:59.451Z"})
```

- Flexibility in querying

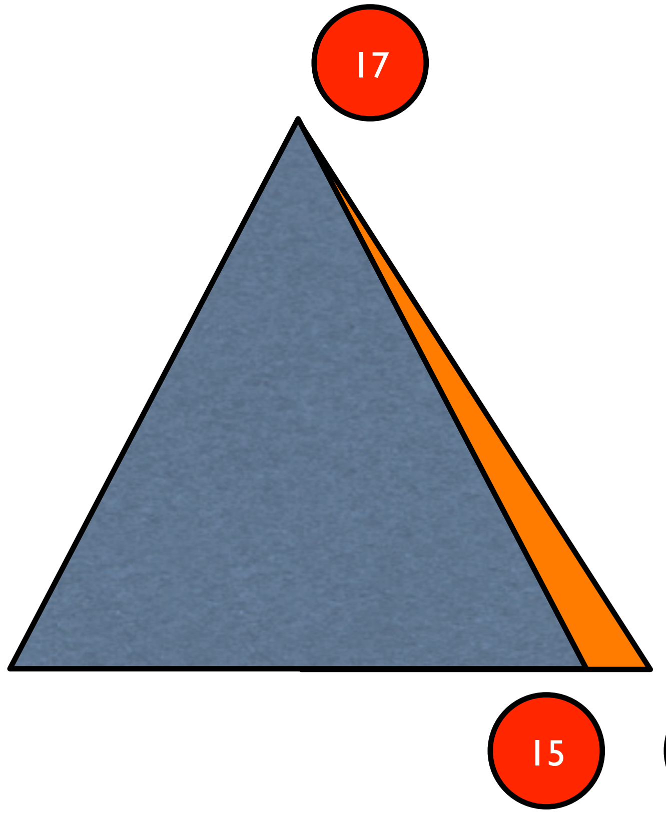


```
// October and later
> db.friendly_lookups.find({"_id": /^2011-10-/})

  specific date/hour
        okups.find({"_id": /^2011-11-27T01:/})
```

Case-
sensitive
REGEX is
your pal

Making the most of _id

- Hot indexes are happy indexes
  - Access should strive for right bias
- Random access with large indexes hit disk



Your Data Model

- NoSQL gets you started *faster*
- Many relational pain points are *gone*
- New considerations (easier?)
- Migration should be real effort
- Designed by *access patterns* over object structure
- Don’t prematurely optimize, but know where the knobs are

More Reading

- http://tech.wordnik.com
- http://github.com/wordnik/wordnik-oss
- http://developer.wordnik.com
- http://slideshare.net/fehguy