Querying Databases: The Non-relational Ways

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Explain why querying non-relational databases is different.
- Identify issues we need to consider for querying non-relational databases.

The Rise of Non-relational DBMS

Handling Data Everywhere → NoSQL

- New kinds of data: Web data, social networks, scientific data.
- New requirements
  - Volume → Scalability
    - Handling extremely large data.
    - Handling extremely many users.
  - Variety → One model may not fit all
    - Handling very simple to very complex data.
- NoSQL databases
  - Originally "non SQL" or "non relational".
  - Now "not only SQL".

NoSQL Data Models

- Key-Value Model
  - Berkeley DB, Redis
- Document Model
  - MongoDB, CouchDB
  - JSON is a popular document model.
- Graph Model
  - Neo4j, OrientDB

Implications of Physical Data Models: *How to Query Non-relations?*

We get results by "assembling" answers from tables.

```
SELECT beer, AVG(price)
FROM Beers, Sells
WHERE Bars.name = Sells.beer
      and brewer = "AB InBev"
GROUP BY beer
HAVING COUNT(bar) >= 2
```

Drinkers				
name	addr	hobby	bar	beer

Bars			
name	addr	owner	beer

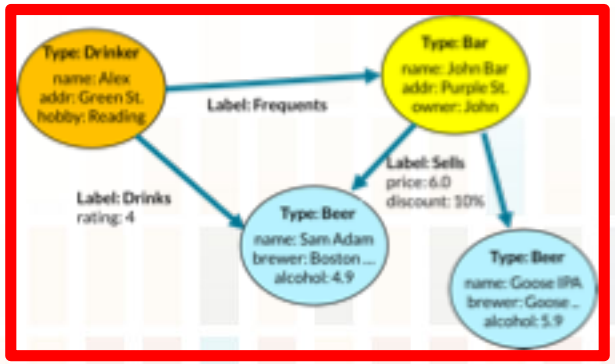
Beers		
name	brewer	alcohol

Frequents	
drinker	bar

• • •

Now, do we assemble "documents"?

What do we assemble over a graph?



Contrasting non-relational models with relations

Querying Relations: Key Concepts


```
SELECT beer, AVG(price) AS AveragePrice
FROM Beers, Sells
WHERE Bars.name = Sells.beer
      and brewer = "AB InBev"
GROUP BY beer
HAVING COUNT(bar) >= 2
```

Concept	Operated Upon	Examples
Reducing	... tuples and attributes.	SELECT beer WHERE brewer = "AB InBev"
Combining	… two relations.	FROM Beers, Sells
Grouping	… tuples by attributes.	GROUP BY beer
Aggregating	… attributes across tuples.	COUNT(bar)
Transforming	… attributes for result tuples.	AVG(price) as AveragePrice

How to Query Non-relational Data?

- What are the key concepts?
- How are they performed?
- Criteria to examine
  - Are they easy to use?
  - Are they powerful to get information we desire?
- Perspective: How are they different from relational querying?

Querying Document Databases: MongoDB

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor
Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Describe the basic mechanism for querying MongoDB.
- Identify the methods used for querying MongoDB.
- Explain how MongoDB querying is conceptually both distinct from and related to relational querying.

Concept Mapping:
From Relations to Documents

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document or BSON document
column	field
index	index
table joins	`$lookup`, embedded documents
primary key	primary key

SQL to MongoDB Mapping Chart. Retrieved from https://docs.mongodb.com/manual/reference/sql-comparison/#sql-to-mongodb-mapping-chart

Querying MongoDB: Mechanism

- Queries are issued using “shell commands”.
- You can access databases using an interactive “mongo Shell”.
- Applications can access databases using “shell methods”.
  - Via client language drivers.

```
> show databases
admin  0.000GB
local  0.000GB
myDB   0.000GB
> use myDB
switched to db myDB
> show collections
Bars
Beers
CompleteBars
CompleteDrinkers
CompleteRefDrinkers
Drinkers
Drinks
Favorites
Sells
> db.Bars.find()
{ "_id" : ObjectId("59e9a24eeabd8c0d2c5dd16c"), "name" : "Sober Bar", "addr" : "Purple St", "owner" : "Jim" }
{ "_id" : ObjectId("59e9a24eeabd8c0d2c5dd16d"), "name" : "Green Bar", "addr" : "Green St", "owner" : "Sally" }
{ "_id" : ObjectId("59e9a24eeabd8c0d2c5dd16e"), "name" : "Purple Bar", "addr" : "Purple St", "owner" : "Paul" }
> db.Beers.find()
{ "_id" : ObjectId("59e83bd5ca5f4f41da44a53b"), "name" : "Sam Adams", "brewer" : "Boston Beer", "alcohol" : 4.9 }
{ "_id" : ObjectId("59e83bd5ca5f4f41da44a53c"), "name" : "Bud", "brewer" : "AB InBev", "alcohol" : 5 }
{ "_id" : ObjectId("59e83bd5ca5f4f41da44a53d"), "name" : "Bud Lite", "brewer" : "AB InBev", "alcohol" : 4.2 }
{ "_id" : ObjectId("59e83bd5ca5f4f41da44a53e"), "name" : "Coors", "brewer" : "Coors", "alcohol" : 5 }
> db.Drinkers.find()
{ "_id" : ObjectId("59e83bd5ca5f4f41da44a535"), "name" : "Alex", "addr" : "Green St", "hobby" : "Reading", "frequents" : "Sober Bar" }
{ "_id" : ObjectId("59e83bd5ca5f4f41da44a536"), "name" : "Betty", "addr" : "King St", "hobby" : "Singing", "frequents" : "Green Bar" }
{ "_id" : ObjectId("59e83bd5ca5f4f41da44a537"), "name" : "Cindy", "addr" : "Green St", "hobby" : "Hiking", "frequents" : "Green Bar" }
>
```

Accessing MongoDB from mongo Shell

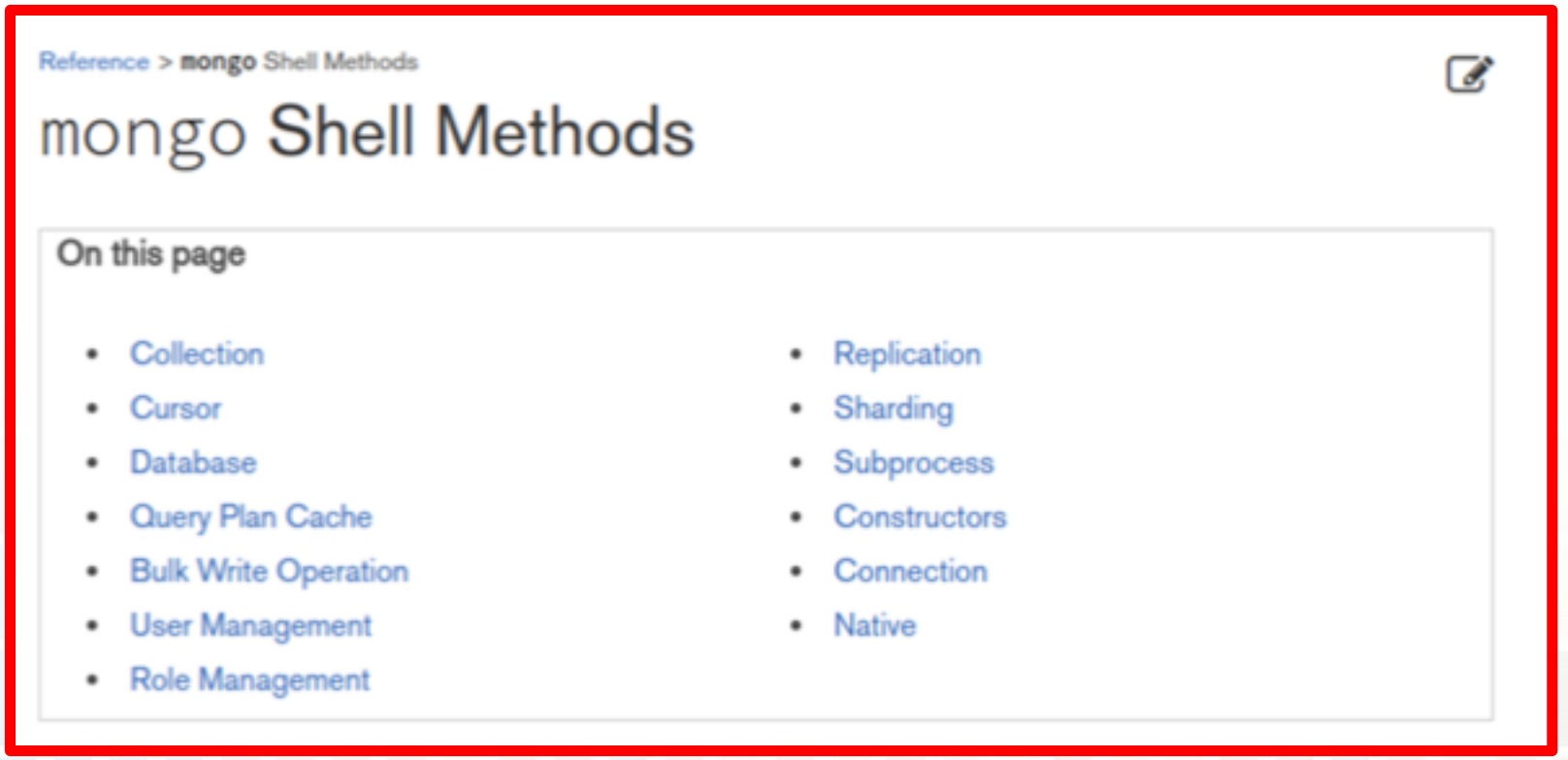
Accessing MongoDB from Python

MongoDB Querying Examples

- *Q1: Using mongo Shell , explore the FridayNight database.*
- *Q2: Perform some querying from Python.*

Mongo Shell Methods

- A set of methods to query and update data as well as perform administrative operations.



MongoDB documentation, 2017. Retrieved from https://docs.mongodb.com/manual/reference/method/

Querying MongoDB: Shell Methods

Method	Description
db.collection.aggregate()	Provides access to the aggregation pipeline.
db.collection.count()	Return a count of the number of documents in a collection or a view.
db.collection.distinct()	Returns an array of documents that have distinct values for the specified field.
db.collection.find()	Performs a query on a collection or a view and returns a cursor object.
db.collection.findOne()	Performs a query and returns a single document.
db.collection.mapReduce()	Performs map-reduce style data aggregation.

Doc Querying: *Collection* Perspective

- All query methods are **collection** methods.
- What this means? A query is performed in either forms:
  - Operating within the scope of one collection.
  - Operating from one primary collection to another (asymmetrical).

Method	Description
db.collection.aggregate()	Provides access to the aggregation pipeline.
db.collection.count()	Return a count of the number of documents in a collection or a view.
db.collection.distinct()	Returns an array of documents that have distinct values for the specified field.
db.collection.find()	Performs a query on a collection or a view and returns a cursor object.
db.collection.findOne()	Performs a query and returns a single document.
db.collection.mapReduce()	Performs map-reduce style data aggregation.

Doc Querying: "*Relational/SQL*" Capabilities

- Much influence from relational/SQL capabilities
  - Evident from the design, documentation, and explanation throughout.
- That's why we study "the relational way" of querying.

MongoDB documentation, 2017. Retrieved from https://docs.mongodb.com/manual/reference/operator/query/

SQL Terms, Functions, and Concepts	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum
join	$lookup

MongoDB documentation, 2017. Retrieved from https://docs.mongodb.com/manual/reference/sql-aggregation-comparison/

MongoDB blog, Andrew Morgan, 10/20/2015. Retrieved from https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-1-of-3-introduction

Querying Document Databases: Basic Operations

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

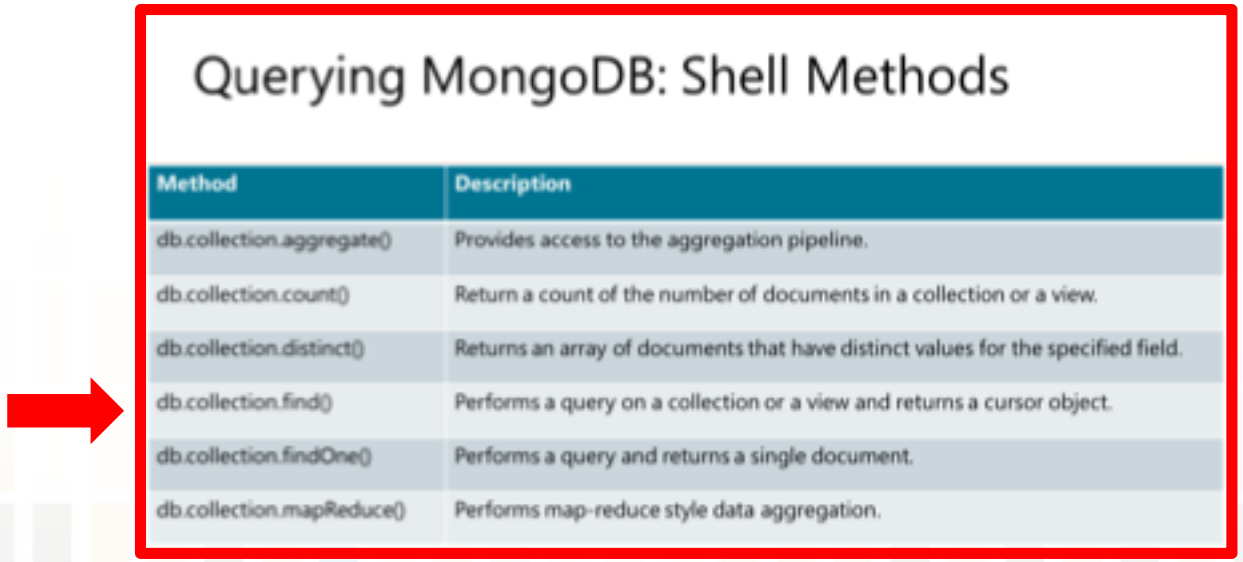
Learning Objectives

By the end of this video, you will be able to:

- Describe the form and functions of the basic query operation db.collection.find.
- Contrast the capabilities of the find method to relational operations.
- Explain why document databases do not generally support joins.
- Write queries with basic operations.

The Basic Method: db.collection.find()

- MongoDB supports querying through several shell methods.
- Not totally complementary-- quite some overlapping.
- **db.collection.find()** is the basic.



Querying MongoDB: Shell Methods

Method	Description
db.collection.aggregate()	Provides access to the aggregation pipeline.
db.collection.count()	Return a count of the number of documents in a collection or a view.
db.collection.distinct()	Returns an array of documents that have distinct values for the specified field.
db.collection.find()	Performs a query on a collection or a view and returns a cursor object.
db.collection.findOne()	Performs a query and returns a single document.
db.collection.mapReduce()	Performs map-reduce style data aggregation.

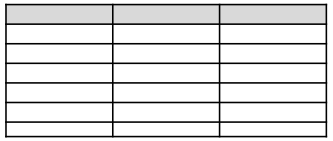
We Will Contrast with Relational Operations

Basic Operators

- Reduction: *Make a table smaller.*
  - Selection σ
  - Projection π

What is the major of Bugs Bunny?

Students

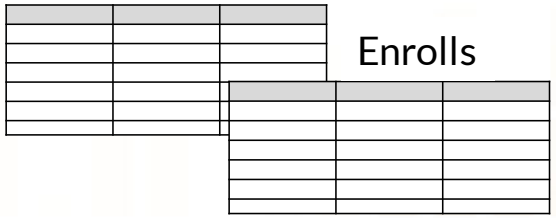


- Combination: *Combine two tables.*
  - Set Union $\cup$
  - Set Difference $-$
  - Cartesian Product $\times$

What courses are Bugs Bunny taking?

Students

Enrolls



- Renaming ρ: *Change attribute names.*

Querying a Collection: Find()

`db.collection.find(query, projection)`

Selects documents in a collection or view and returns a cursor to the selected documents.

Parameter	Type	Description
`query`	document	Optional. Specifies selection filter using query operators. To return all documents in a collection, omit this parameter or pass an empty document ({}).
`projection`	document	Optional. Specifies the fields to return in the documents that match the query filter. To return all fields in the matching documents, omit this parameter. For details, see Projection.

Returns: A cursor to the documents that match the `query` criteria. When the `find()` method "returns documents," the method is actually returning a cursor to the documents.

db.collection.find() documentation. Retrieved from https://docs.mongodb.com/manual/reference/method/db.collection.find/

Basic Query Examples

- *Q1: Find the beers that are brewed by AB InBev.*
- *Q2: Find the beers that are available at a price less than $5.*

Binary Operations:
Union, Difference, Cartesian Product (Join)

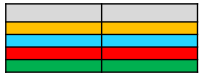
- No way to perform operations over collections.
- There are "similar operations" (e.g., $setUnion) for combining results within a collection.

Combination Operators

- Combining two relations R_1 and R_2 :

Two relations

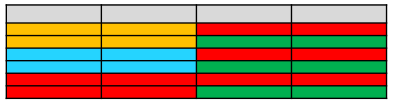
- Set operations
  - Union: $R_1 \cup R_2$ (addition)
  - Difference: $R_1 - R_2$ (like subtraction)



Union

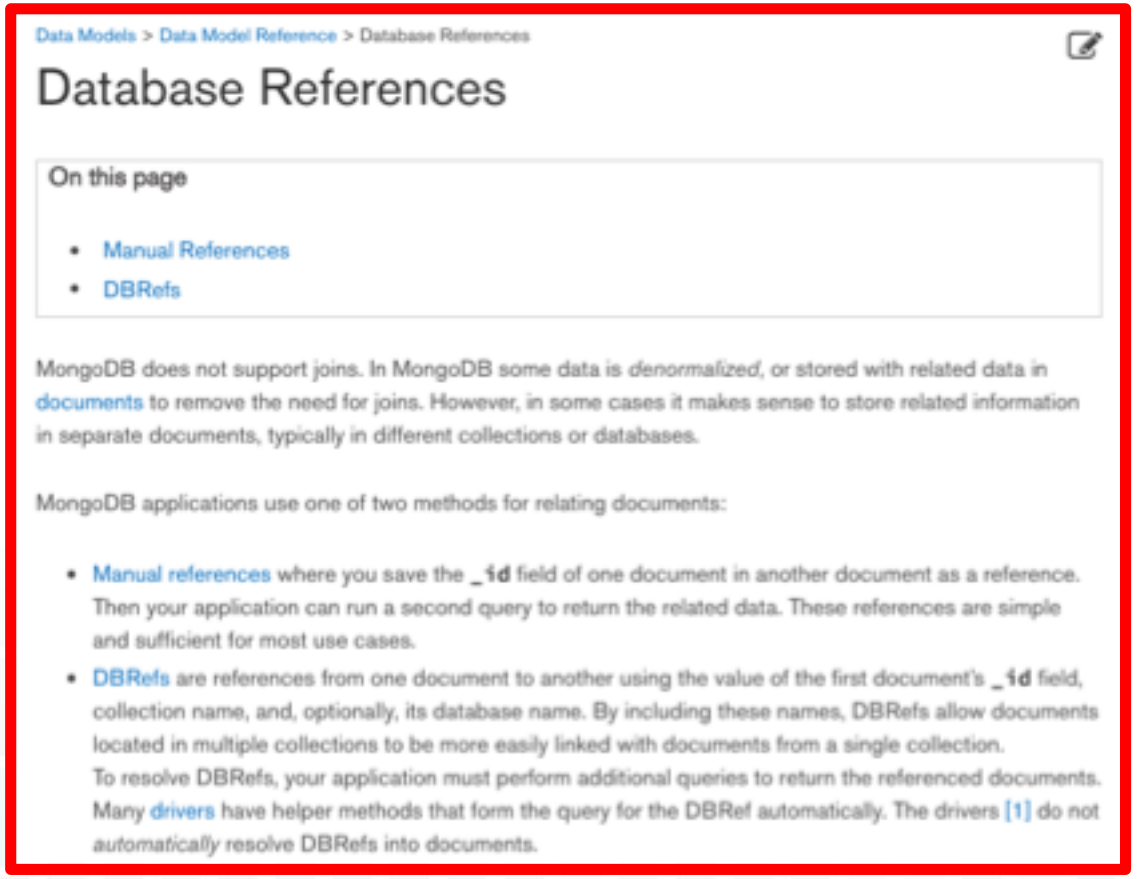
Difference

- Cartesian product (multiplication)
  - $R_1 \times R_2$



Cartesian product

Why Not Joins or Other Binary Ops?



Data Models > Data Model Reference > Database References

Database References

On this page

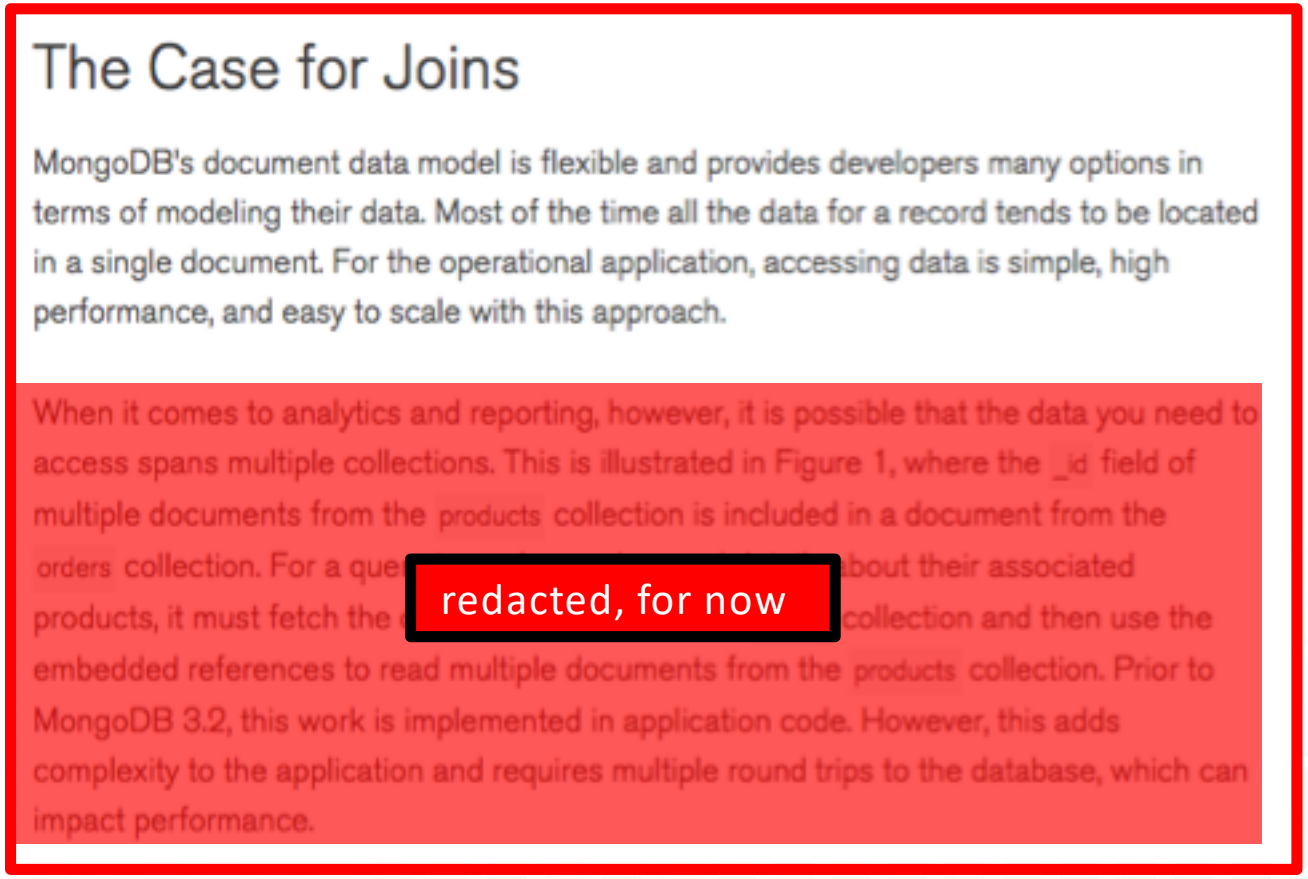
- Manual References
- DBRefs

MongoDB does not support joins. In MongoDB some data is *denormalized*, or stored with related data in documents to remove the need for joins. However, in some cases it makes sense to store related information in separate documents, typically in different collections or databases.

MongoDB applications use one of two methods for relating documents:

- Manual references where you save the **_id** field of one document in another document as a reference. Then your application can run a second query to return the related data. These references are simple and sufficient for most use cases.
- DBRefs are references from one document to another using the value of the first document's **_id** field, collection name, and, optionally, its database name. By including these names, DBRefs allow documents located in multiple collections to be more easily linked with documents from a single collection. To resolve DBRefs, your application must perform additional queries to return the referenced documents. Many drivers have helper methods that form the query for the DBRef automatically. The drivers [1] do not *automatically* resolve DBRefs into documents.

Database References, 2017. Retrieved from https://docs.mongodb.com/manual/reference/database-references/



The Case for Joins

MongoDB's document data model is flexible and provides developers many options in terms of modeling their data. Most of the time all the data for a record tends to be located in a single document. For the operational application, accessing data is simple, high performance, and easy to scale with this approach.

When it comes to analytics and reporting, however, it is possible that the data you need to access spans multiple collections. This is illustrated in Figure 1, where the _id field of multiple documents from the products collection is included in a document from the orders collection. For a que bout their associated products, it must fetch the collection and then use the embedded references to read multiple documents from the products collection. Prior to MongoDB 3.2, this work is implemented in application code. However, this adds complexity to the application and requires multiple round trips to the database, which can impact performance.

redacted, for now

Database References, 2017. Retrieved from https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-1-of-3-introduction

Binary Operations: Do It Yourself!

- $\pi_{\text{name}}\sigma_{\text{brewer}=\text{“AB InBev"}}\textbf{Beers} \cap \pi_{\text{name}}\ \sigma_{\text{price}<5.0}\textbf{Sells}$

```
# MongoDB driver
import pymongo

# Making connection to MongoDB
conn=pymongo.MongoClient()
db = conn.CS411

Q1 = db.Beers.find({"brewer":"AB InBev"}, {"name":1})
Q2 = db.Sells.find({"price":{"$lt":5}}, {"beer":1})

print ("** Results of Q1:")
for d1 in Q1:
    name = d1["name"]
    print name

print ("** Results of Q2:")
for d2 in Q2:
    beer = d2["beer"]
    print beer

results = []
print ("** Intersection:")
for d1 in Q1.rewind():
    for d2 in Q2.rewind():
        if (d1["name"] == d2["beer"] and d1["name"] not in results):
            results.insert(0, d1["name"])
            print d1["name"]
```

Performing intersection of two queries in Python

```
$ python intersection.py
** Results of Q1:
Bud
Bud Lite
** Results of Q2:
Bud Lite
Sam Adams
** Intersection:
Bud Lite
```

Results of intersection

Binary Operation Examples

- *Q1: Find beers brewed by AB InBev AND is available at price less than $5.*

- *Q2: Find beers brewed by AB InBev OR is available at price less than $5.*

Renaming

- Not supported in the basic find() command.
- However, supported in a similar construct in Aggregate.

The Find() Method Compared

- db.collection.find(**selection, projection**)

Relation Operation	Supported in db.collection.Find()
Selection	Yes
Projection	Yes
Set Union	No
Set Difference	No
Cartesian Product	No
Renaming	No

Querying Document Databases: Handling Complex Structures

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

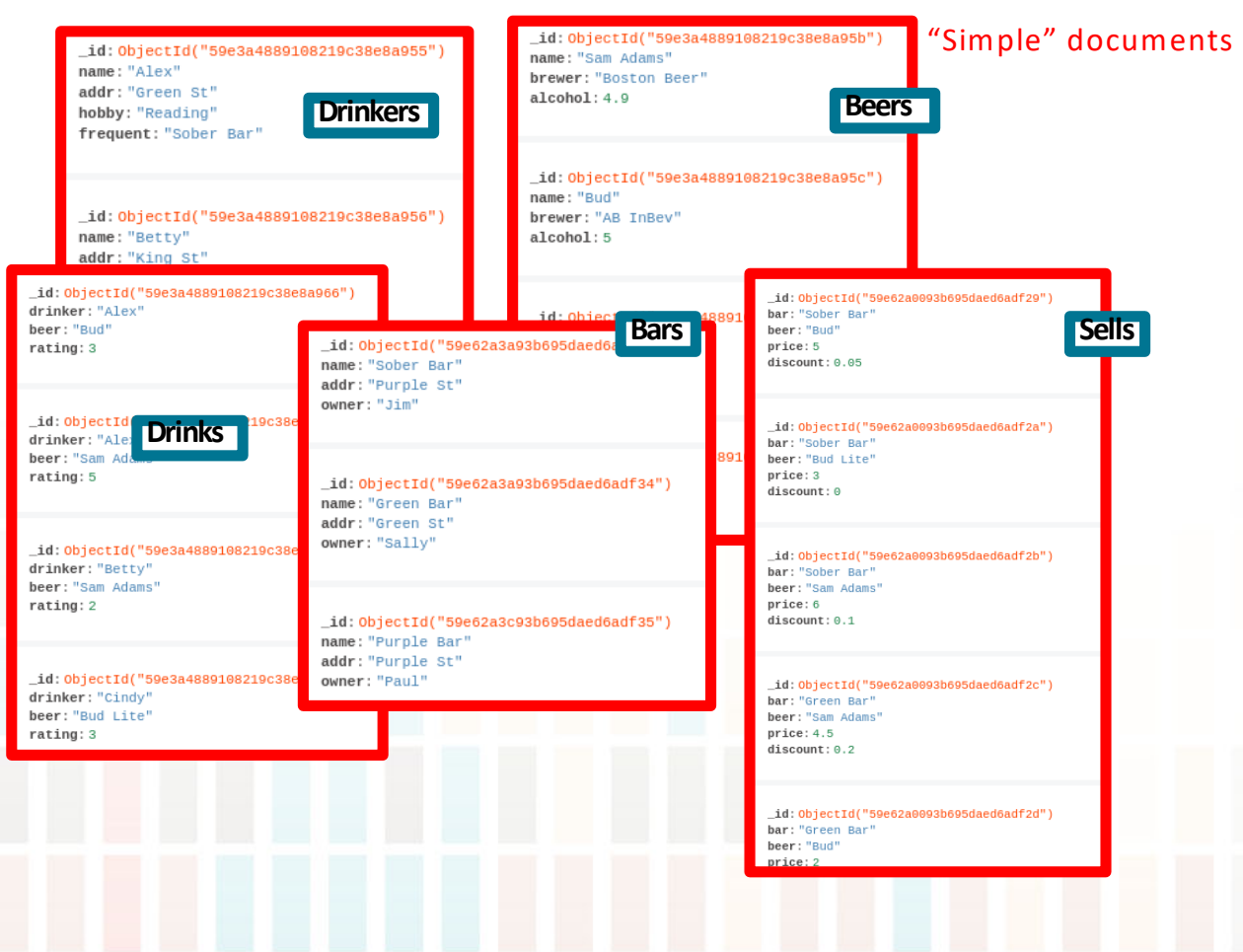
Learning Objectives

By the end of this video, you will be able to:

- Identify ways to deal with complex structures of documents in MongoDB querying.
- Describe different options for forming complex documents and compare how they are handled in querying.
- Write queries involving complex documents.

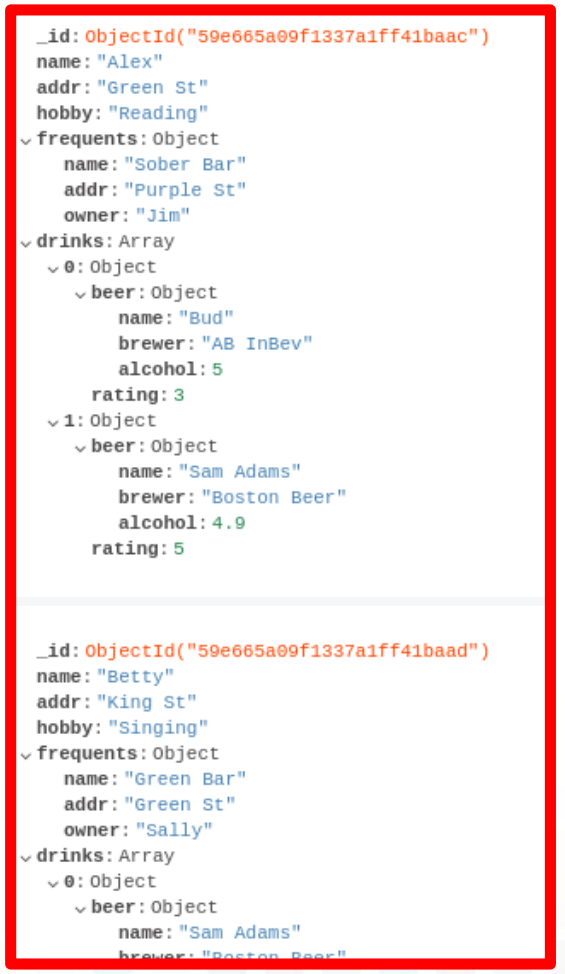
Dealing with Complex Structures

- We learned to “normalize” in the relational structure.
- The document model assumes “denormalized” document structure.



Querying Embedded Documents¶

- Condition = Attribute Operator Value.
- Complex values
  - *Q: Find drinkers who frequent a bar with name X, address Y, and owner Z.*
  - db.CompleteDrinkers.find({
    frequents:{name:"Green Bar", addr:"Green St", owner:"Sally"}}, {name:1})
- Complex attributes
  - *Q: Find drinkers who frequent a bar at address Y.*
  - db.CompleteDrinkers.find({"frequents.addr":"Green St"}, {name:1})
- Complex operators
  - *Q: Find drinkers who drink all these beers A, B, and C.*
  - db.CompleteDrinkers.find({"drinks.beer.name": {$all:["Bud", "Sam Adams"]}}, {name:1})



Complex-Structure Query Examples

- *Q1: Find the drinkers who frequent a bar with name X, address Y, and owner Z (over CompleteDrinkers collection).*
- *Q2: Find the drinkers who frequent a bar at address Y.*
- *Q3: Find the drinkers who drink all these beers A, B, and C.*

Complex Structure:
Choices of Embedding or References

CompleteDrinkers

Embedding

CompleteRefDrinkers

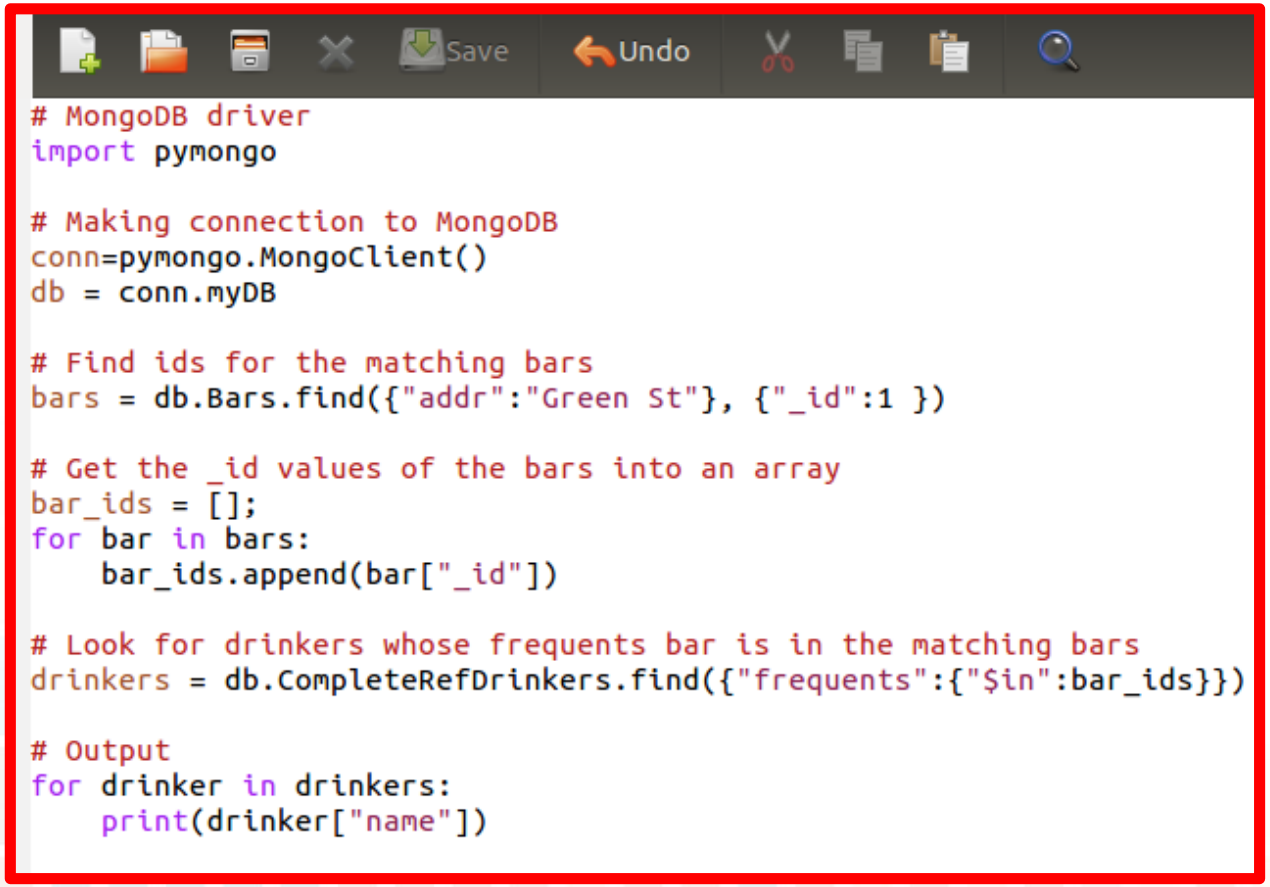
Referencing

Querying into Referenced Documents

- *Q: Drinkers who frequent a bar at address Y?*
- db.CompleteDrinkers.find({"frequents.addr":"Green St"})?

Querying into Referenced Documents – Doing It Yourself!

- Retrieve referenced objects. Check if match conditions.



Code adapted from https://dba.stackexchange.com/questions/107101/mongodb-querying-for-a-document-that-has-an-object-reference

Referenced-Document Query Examples

- *Q: Find the drinkers who frequent a bar at “Green St” (over CompleteRefDrinkers collection).*

Food for Thought

Using document databases, any reason you would choose object referencing over embedding?

Querying Document Databases: Aggregates

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Describe the “pipeline” framework of aggregate queries in MongoDB.
- Identify the operators used for a stage in the pipeline framework.
- Visualize and explain how the pipeline framework works.
- Write aggregate queries.

Aggregation Framework: Pipeline of Stages

`db.collection.aggregate(pipeline, options)`

Calculates aggregate values for the data in a collection or a view.

Parameter	Type	Description
`pipeline`	array	A sequence of data aggregation operations or stages. See the aggregation pipeline operators for details. *Changed in version 2.6:* The method can still accept the pipeline stages as separate arguments instead of as elements in an array; however, if you do not specify the **`pipeline`** as an array, you cannot specify the **`options`** parameter.
`options`	document	Optional. Additional options that **`aggregate()`** passes to the `aggregate` command.

db.collection.aggregate(). Retrieved from https://docs.mongodb.com/manual/reference/method/db.collection.aggregate/

Pipeline Aggregation Stages

Operator	Relational/SQL Operation	MongoDB Description
$match	selection	Filters the document stream to allow only matching documents to pass unmodified into the next pipeline stage. $match uses standard MongoDB queries. For each input document, outputs either one document (a match) or zero documents (no match).
$project	projection	Reshapes each document in the stream, such as by adding new fields or removing existing fields. For each input document, outputs one document.
$group	group-by	Groups input documents by a specified identifier expression and applies the accumulator expression(s), if specified, to each group. Consumes all input documents and outputs one document per each distinct group. The output documents only contain the identifier field and, if specified, accumulated fields.
$unwind	ungroup-by	Deconstructs an array field from the input documents to output a document for each element. Each output document replaces the array with an element value. For each input document, outputs n documents where n is the number of array elements and can be zero for an empty array.
$lookup	outer join	Reorders the document stream by a specified sort key. Only the order changes; the documents remain unmodified. For each input document, outputs one document.

Aggregate: Subsumes and Generalizes Find()

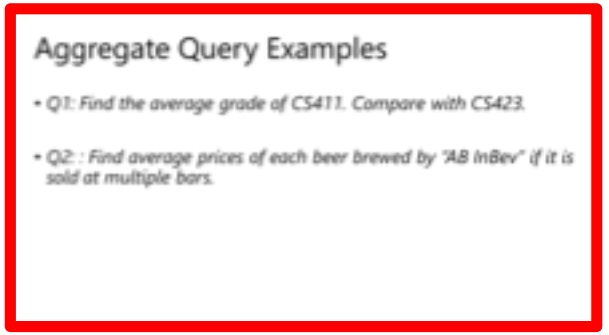
- Simple two-stage pipeline:

  db.collection.aggregate([

  {$match: **{conditions}** }, { $project: **{attributes}** }])

- Similar to db.collection.find(**{conditions}** , **{attributes}**)
  - db.Beers.find({brewer:"AB InBev"}, {"_id":0, name:1})
  - db.Beers.aggregate([
    {$match: {brewer:"AB InBev"}}, {$project: {"_id":0, name:1}}])
- But more general/powerful
  - $project supports renaming, transformation.
  - db.Beers.aggregate([
    {$match: {brewer:"AB InBev"}}, {$project: {"_id":0, beer:"$name"}}])

Aggregate Query Examples

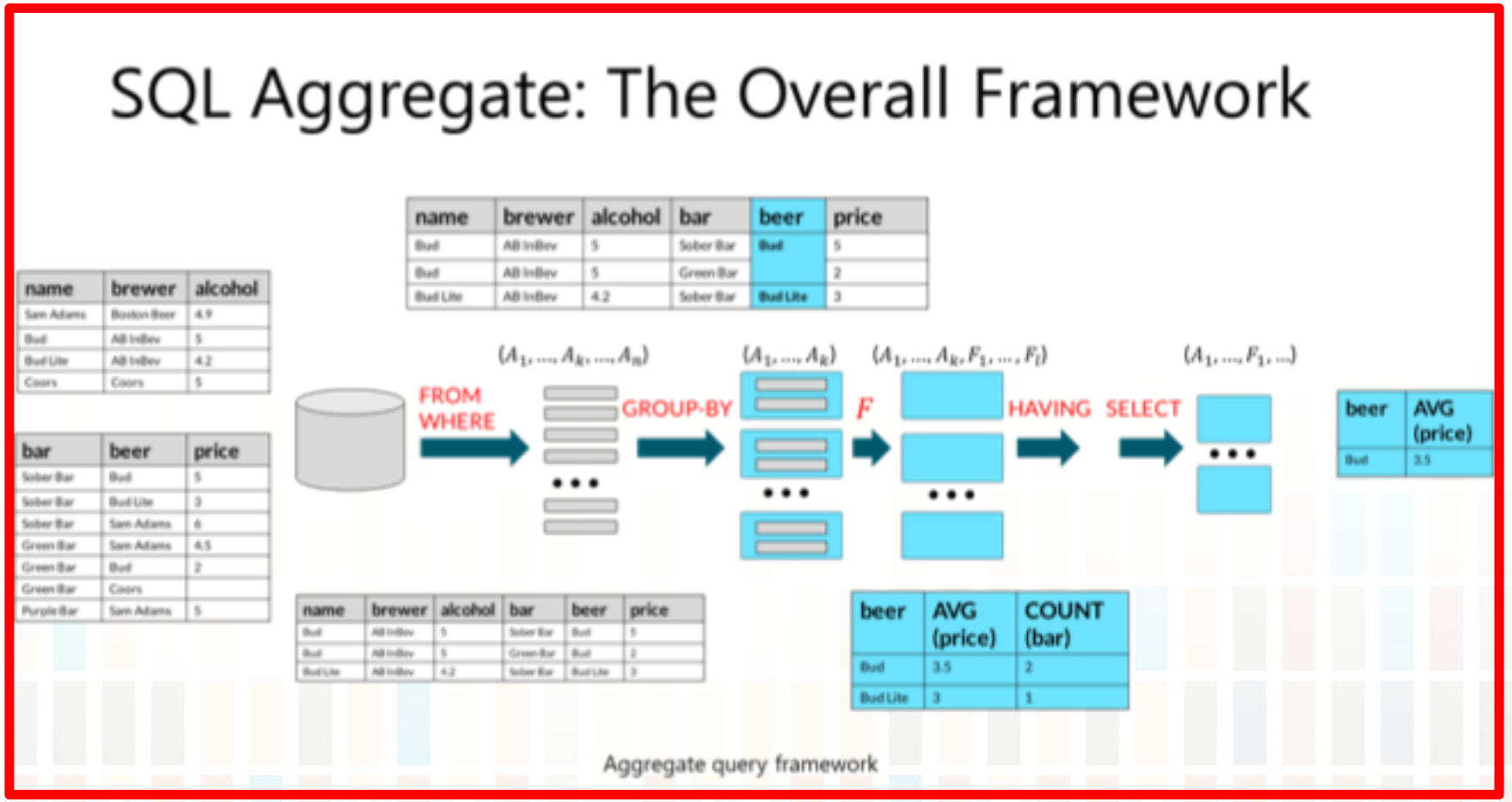


- *Q1: Find the beers brewed by AB InBev.*
- *Q2: Find the average ratings of beers.*
- *Q3: : Find the average price of each beer brewed by "AB InBev" if it is sold at multiple bars.*

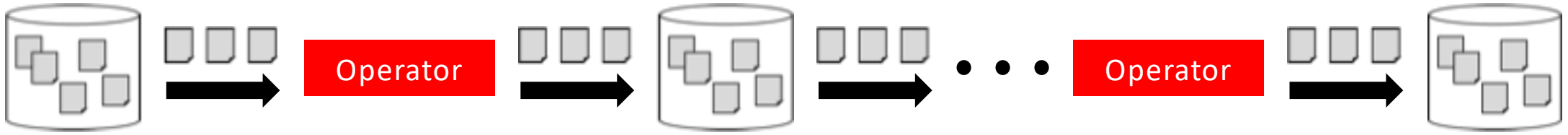
Let's See How It Works for the Same Query!

Q: Find the average price of each beer brewed by "AB InBev" if it is sold at multiple bars.

```
SELECT beer, AVG(price)
FROM Beers, Sells
WHERE Beers.name = Sells.beer
      and brewer = "AB InBev"
GROUP BY beer
HAVING COUNT(bar) >= 2
```



The Pipeline Framework




```
[
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "sells": [
      {
        "beer": {
          "name": "Bud",
          "brewer": "AB InBev",
          "alcohol": 5
        },
        "price": 5,
        "discount": 0.05
      },
      {
        "beer": {
          "name": "Bud Lite",
          "brewer": "AB InBev",
          "alcohol": 4.2
        },
        "price": 3,
        "discount": 0
      },
      {
        "beer": {
          "name": "Sam Adams",
          "brewer": "Boston Beer",
          "alcohol": 4.9
        },
        "price": 6,
        "discount": 0.1
      }
    ]
  },
  {
    "name": "Green Bar",
    "addr": "Green St",
    "owner": "Sally",
    "sells": [
      {
        "beer": {
          "name": "Sam Adams",
          "brewer": "Boston Beer",
          "alcohol": 4.9
        },
        "price": 4.5,
        "discount": 0.2
```


$unwind

$match

```
db.CompleteBars.aggregate([
        {$unwind:"$sells"},
        {$match:{"sells.beer.brewer":"AB InBev"}},
        {$group:{_id:"$sells.beer.name",
         countSales:{$sum:1},
         avgPrice:{$avg:"$sells.price"}}},
        {$match:{"countSales":{$gt:1}}}
])
```


```
[
  {
    "_id": "Bud",
    "countSales": 3,
    "avgPrice": 3.5
  }
]
```


The pipeline framework for aggregates in MongoDB

Aggregate Pipeline #1: $unwind

```
db.CompleteBars.aggregate([
        {$unwind:"$sells"},
        {$match:{"sells.beer.brewer":"AB InBev"}},
        {$group:{_id:"$sells.beer.name",
            countSales:{$sum:1},
            avgPrice:{$avg:"$sells.price"}}},
        {$match:{"countSales":{$gt:1}}}
])
```


```
[
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "sells": [
      {
        "beer": {
          "name": "Bud",
          "brewer": "AB InBev",
          "alcohol": 5
        },
        "price": 5,
        "discount": 0.05
      },
      {
        "beer": {
          "name": "Bud Lite",
          "brewer": "AB InBev",
          "alcohol": 4.2
        },
        "price": 3,
        "discount": 0
      },
      {
        "beer": {
          "name": "Sam Adams",
          "brewer": "Boston Beer",
          "alcohol": 4.9
        },
        "price": 6,
        "discount": 0.1
      }
    ]
  },
  {
    "name": "Green Bar",
    "addr": "Green St",
    "owner": "Sally",
    "sells": [
      {
        "beer": {
          "name": "Sam Adams",
          "brewer": "Boston Beer",
          "alcohol": 4.9
        },
        "price": 4.5,
        "discount": 0.2
```


$unwind


```
[
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "sells": {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 5,
      "discount": 0.05
    }
  },
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "sells": {
      "beer": {
        "name": "Bud Lite",
        "brewer": "AB InBev",
        "alcohol": 4.2
      },
      "price": 3,
      "discount": 0
    }
  },
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "sells": {
      "beer": {
        "name": "Sam Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.9
      },
      "price": 6,
      "discount": 0.1
    }
  },
  {
    "name": "Green Bar",
    "addr": "Green St",
    "owner": "Sally",
    "sells": {
```

Aggregate Pipeline #2: $match

```
db.CompleteBars.aggregate([
        {$unwind:"$sells"},
        {$match:{"sells.beer.brewer":"AB InBev"}},
        {$group:{_id:"$sells.beer.name",
            countSales:{$sum:1},
            avgPrice:{$avg:"$sells.price"}}},
        {$match:{"countSales":{$gt:1}}}
])
```

$unwind

$match

Aggregate Pipeline #3: $group

```
db.CompleteBars.aggregate([
        {$unwind:"$sells"},
        {$match:{"sells.beer.brewer":"AB InBev"}},
        {$group:{_id:"$sells.beer.name",
              countSales:{$sum:1},
              avgPrice:{$avg:"$sells.price"}}},
        {$match:{"countSales":{$gt:1}}}
])
```


```
[
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "sells": {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 5,
      "discount": 0.05
    }
  },
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "sells": {
      "beer": {
        "name": "Bud Lite",
        "brewer": "AB InBev",
        "alcohol": 4.2
      },
      "price": 3,
      "discount": 0
    }
  },
  {
    "name": "Green Bar",
    "addr": "Green St",
    "owner": "Sally",
    "sells": {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 2,
      "discount": 0
    }
  },
  {
    "name": "Purple Bar",
    "addr": "Purple St",
    "owner": "Paul",
    "sells": {
```


$group


```
[
  {
    "_id": "Bud Lite",
    "countSales": 1,
    "avgPrice": 3
  },
  {
    "_id": "Bud",
    "countSales": 3,
    "avgPrice": 3.5
  }
]
```

Aggregate Pipeline #4: $match (Again!)

```
db.CompleteBars.aggregate([
        {$unwind:"$sells"},
        {$match:{"sells.beer.brewer":"AB InBev"}},
        {$group:{_id:"$sells.beer.name",
            countSales:{$sum:1},
            avgPrice:{$avg:"$sells.price"}}},
        {$match:{"countSales":{$gt:1}}}
])
```

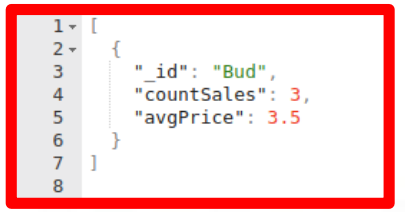
$unwind

$match

$group

```
[
  {
    "_id": "Bud Lite",
    "countSales": 1,
    "avgPrice": 3
  },
  {
    "_id": "Bud",
    "countSales": 3,
    "avgPrice": 3.5
  }
]
```

$match



```
[
  {
    "_id": "Bud",
    "countSales": 3,
    "avgPrice": 3.5
  }
]
```

Food for Thought

How do you compare the pipeline framework of MongoDB aggregates to SQL aggregate queries?

VS

Querying Document Databases: The Case for Joins

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Describe the case for joins– why MongoDB eventually supports joins.
- Explain why MongoDB's join support is different from relational joins and why it is limited.
- Write join queries.

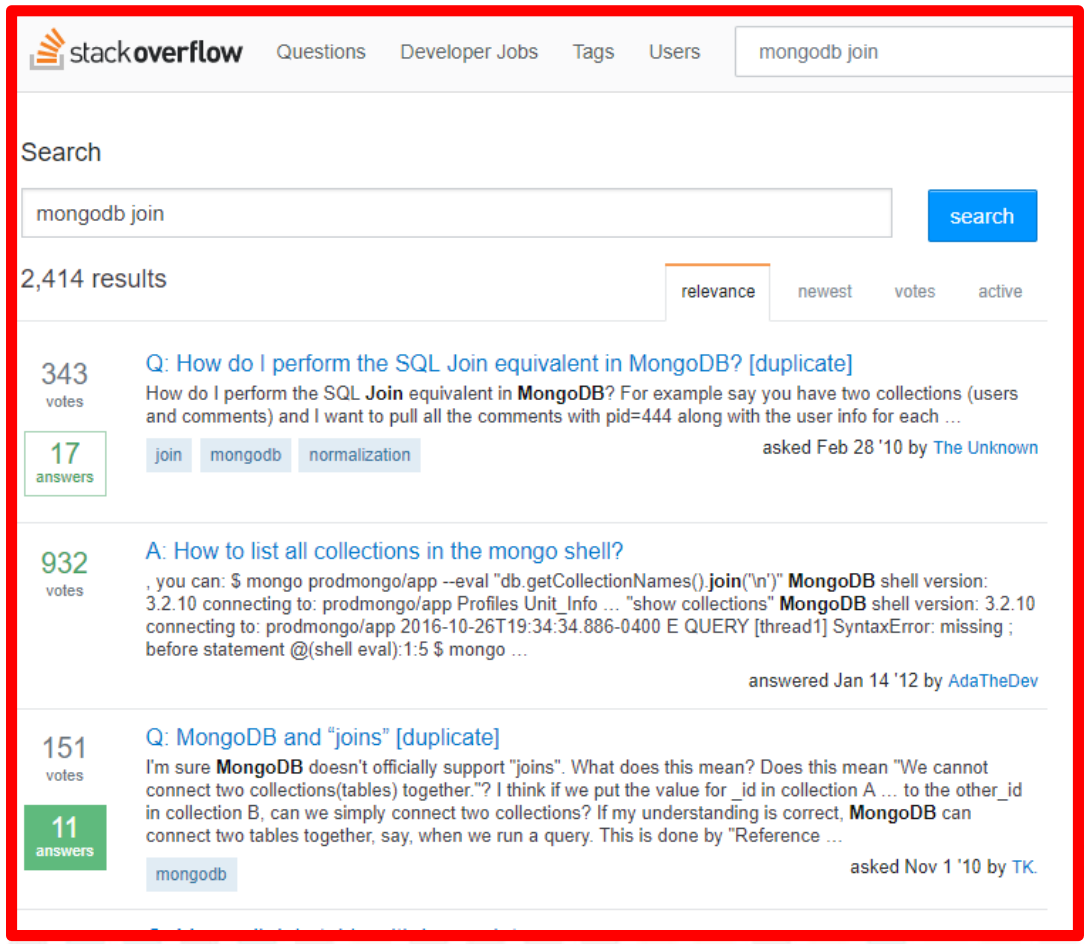
Recall Our “Complete” Drinkers and Bars

Can We Do Away Joins?

Is complex object a panacea?

- You need document at different places.
  - Redundancy!
- There are such places you did not envision.
  - Reorganize your DB!
- And nesting can go arbitrarily deep.
  - Bar – Beer – Brewer – Beer – Bar - …
  - Limits on document size (16MB at MongoDB).

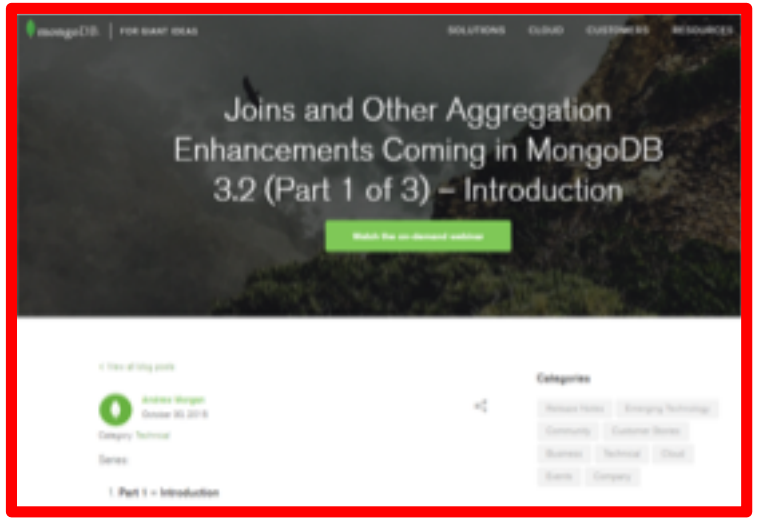
Really?
The Chase for Joins Started Early On!



Search "mongodb join" on StackOverflow, 2017. Retrieved from https://stackoverflow.com/

MongoDB, Wikipedia, 2017. Retrieved from https://en.wikipedia.org/wiki/MongoDB

The Chase is Over. The Case for Joins.



MongoDB blog, Andrew Morgan, 10/20/2015. Retrieved from https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-1-of-3-introduction

To-the-New Blog, 2017. Retrieved from http://www.tothenew.com

The Case for Joins

MongoDB's document data model is flexible and provides developers many options in terms of modeling their data. Most of the time all the data for a record tends to be located in a single document. For the operational application, accessing data is simple, high performance, and easy to scale with this approach.

When it comes to analytics and reporting, however, it is possible that the data you need to access spans multiple collections. This is illustrated in Figure 1, where the `_id` field of multiple documents from the `products` collection is included in a document from the `orders` collection. For a query to analyze orders and details about their associated products, it must fetch the order document from the `orders` collection and then use the embedded references to read multiple documents from the `products` collection. Prior to MongoDB 3.2, this work is implemented in application code. However, this adds complexity to the application and requires multiple round trips to the database, which can impact performance.

Database References, 2017. Retrieved from https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-1-of-3-introduction

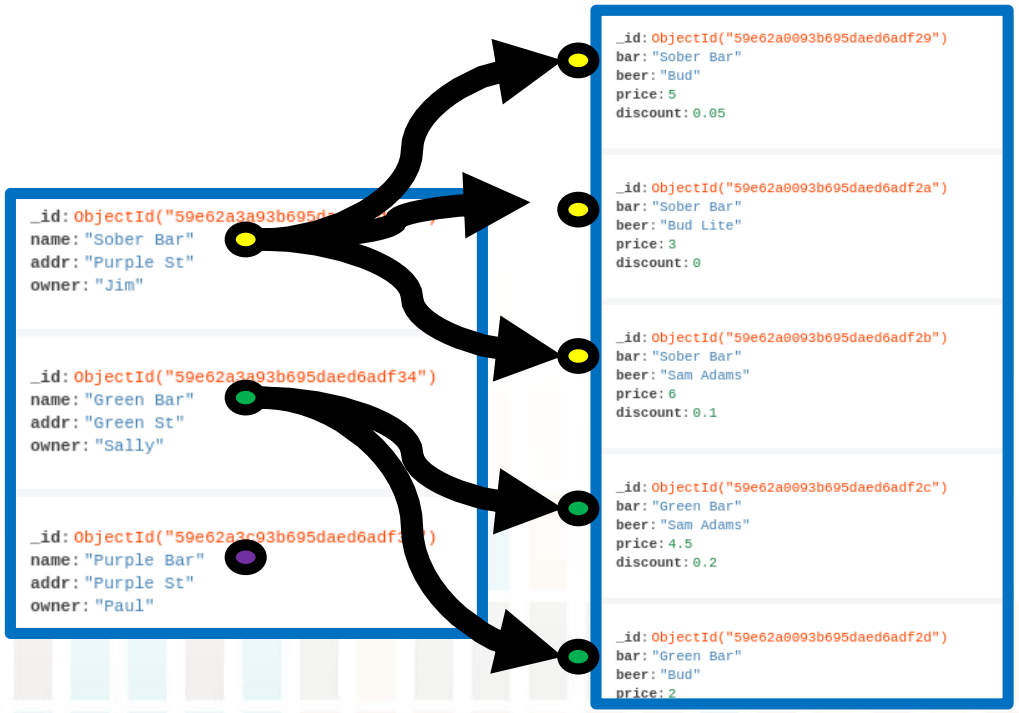
Outer Equi-Join: Collection-Centric Perspective

- For every document in the left collection:
  - Find every document with equi-matching in the right collection.

MongoDB Blog 2017. Retrieved from https://www.mongodb.com/blog/post/joins-and-other-aggregation-enhancements-coming-in-mongodb-3-2-part-1-of-3-introduction

Outer Equi-Join

```
db.Bars.aggregate([
  {$lookup:
      {from:"Sells", localField:"name", foreignField:"bar", as:"menu"}},
  {$project:{"_id":0, "menu._id":0, "menu.bar":0}}
])
```



$lookup

```
[
  {
    "name": "Sober Bar",
    "addr": "Purple St",
    "owner": "Jim",
    "menu": [
      {
        "beer": "Bud",
        "price": 5,
        "discount": 0.05
      },
      {
        "beer": "Bud Lite",
        "price": 3,
        "discount": 0
      },
      {
        "beer": "Sam Adams",
        "price": 6,
        "discount": 0.1
      }
    ]
  },
  {
    "name": "Green Bar",
    "addr": "Green St",
    "owner": "Sally",
    "menu": [
      {
        "beer": "Sam Adams",
        "price": 4.5,
        "discount": 0.2
      },
      {
        "beer": "Coors",
        "price": null,
        "discount": null
      }
```

Outer equi-join by $lookup

Join Query Examples

- *Q1: Find, for each bar, the beers that it sells.*
- *Q2: Find drinker "buddies"– i.e., those who favorite the same beers.*

Food for Thought

Process this query in your mind. What does it do?

```
db.Favorites.aggregate([
{$lookup:{from:"Favorites", localField:"beer", foreignField:"beer", as:"buddy"}},
{$unwind:"$buddy"},
{$project:{drinker:1, "buddy.drinker":1,
   cmp_bars:{$cmp:["$bar", "$buddy.bar"]},
   cmp_drinkers:{$cmp:["$drinker", "$buddy.drinker"]}}},
{$match:{cmp_bars:0, cmp_drinkers:1}},
{$project:{"_id":0, "realBuddy1":"$drinker", "realBuddy2":"$buddy.drinker"}}
])
```

Food for Thought

Since MongoDB can use "aggregate" to support join, it can also support set intersection and union between collections. Agree?

Querying Graph Databases: Neo4j

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

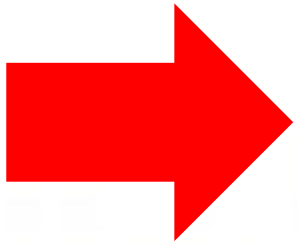
By the end of this video, you will be able to:

- Describe the basic mechanism for querying Neo4j.
- Identify the language used for querying Neo4j and how it is conceptually related to SQL.

FridayNight Database– Now It's a Graph!

Relational database

Document database



Graph database

Neo4j Query Language: Cypher

3.1.1. What is Cypher?

Cypher is a declarative graph query language that allows for expressive and efficient querying and updating of the graph store. Cypher is a relatively simple but still very powerful language. Very complicated database queries can easily be expressed through Cypher. This allows you to focus on your domain instead of getting lost in database access.

Cypher is designed to be a humane query language, suitable for both developers and (importantly, we think) operations professionals. Our guiding goal is to make the simple things easy, and the complex things possible. Its constructs are based on English prose and neat iconography which helps to make queries more self-explanatory. We have tried to optimize the language for reading and not for writing.

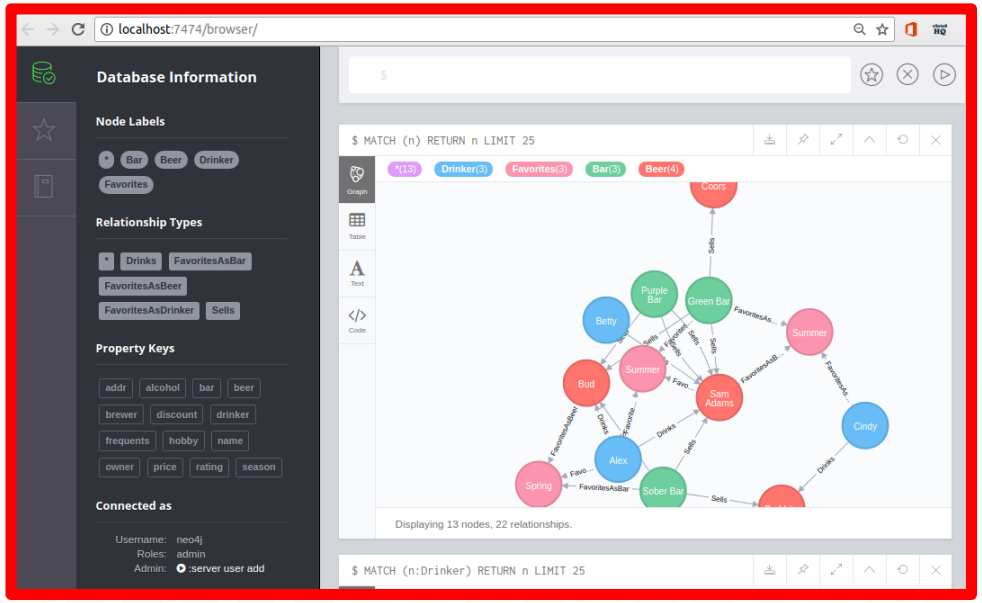
Being a declarative language, Cypher focuses on the clarity of expressing *what* to retrieve from a graph, not on *how* to retrieve it. This is in contrast to imperative languages like Java, scripting languages like Gremlin, and the JRuby Neo4j bindings. This approach makes query optimization an implementation detail instead of burdening the user with it and requiring her to update all traversals just because the physical database structure has changed (new indexes etc.).

Cypher is inspired by a number of different approaches and builds upon established practices for expressive querying. Most of the keywords like `WHERE` and `ORDER BY` are inspired by SQL. Pattern matching borrows expression approaches from SPARQL. Some of the list semantics have been borrowed from languages such as Haskell and Python.

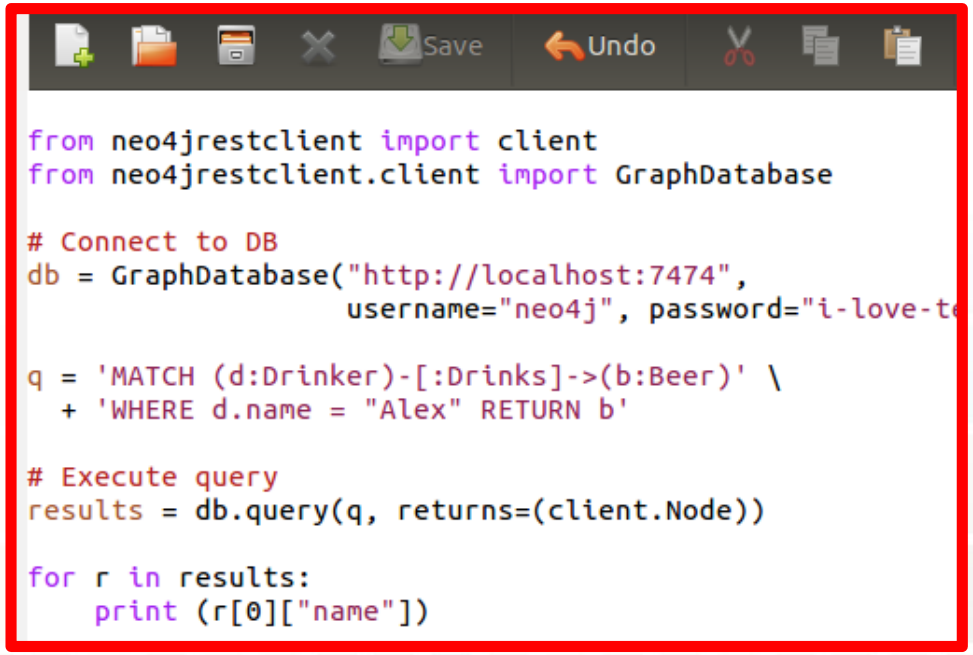
Cypher Manual 2017. Retrieved from https://neo4j.com/docs/developer-manual

Querying Neo4j: Mechanism

- Queries are issued in the Cypher language.
- You can access databases using an interactive GUI shell “Neo4j Browser”.
- Applications can access databases via client language drivers.



Accessing Neo4J from Neo4J Browser



Accessing N4o4J from Python

Neo4j Querying Examples

- *Q1: Using Neo4J Browser, explore the FridayNight database.*
- *Q2: Perform some querying from Python.*

Querying Graph Databases: Basic Operations

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Describe the basic query operations in Neo4j Cypher with MATCH-WHERE-RETURN.
- Explain how these constructs are related to SQL.
- Describe how to use patterns in Neo4j Cypher.
- Write graph queries with simple operations.

Cypher Structure: **MATCH-WHERE-RETURN** vs. SQL: FROM-WHERE-SELECT

- **Structure**

  Cypher borrows its structure from SQL — queries are built up using various clauses.

  Clauses are chained together, and they feed intermediate result sets between each other. For example, the matching variables from one `MATCH` clause will be the context that the next clause exists in.

  The query language is comprised of several distinct clauses. You can read more details about them later in the manual.

  Here are a few clauses used to read from the graph:

  - `MATCH`: The graph pattern to match. This is the most common way to get data from the graph.
  - `WHERE`: Not a clause in its own right, but rather part of `MATCH`, `OPTIONAL MATCH` and `WITH`. Adds constraints to a pattern, or filters the intermediate result passing through `WITH`.
  - `RETURN`: What to return.

Cypher Manual 2017. Retrieved from https://neo4j.com/docs/developer-manual

- Correspondence:

SQL	Neo4j Cypher
FROM	MATCH
WHERE	WHERE
SELECT	RETURN

Patterns

- Nodes: (b:Beer {name:”Bud”})
- Relationships: (bar:Bar)-[s:Sells] -> (beer:Beer)
- Patterns can be written continuously or separated with commas.
  - (bar:Bar)-[s:Sells] -> (beer:Beer)<-[d:Drinks]-(drinker:Drinker)
  - (bar:Bar)-[s:Sells] -> (beer:Beer), (bar:Bar {addr:”Green St”})

- Variables
  - You can refer to variables declared earlier or introduce new ones.
  - You can post conditions on these variables.
    - E.g., b.name = “Bud”.

MATCH-WHERE-RETURN: Over a Node

- A table of an entity becomes a node in a graph DB.
- Querying over a node: **(var: Label)**

- *Q: Find the beers with alcohol greater than 5%.*

SQL	Neo4j Cypher
FROM Beers beer	**MATCH** (beer:Beer)
WHERE beer.alcohol > 0.05	**WHERE** beer.alcohol > 0.05
SELECT beer.name, beer.alcohol	**RETURN** beer.name, beer.alcohol

MATCH-WHERE-RETURN: Over a Relationship

- A table of a relationship becomes, naturally, a relationship in a graph DB.
- Querying over a node: **(…)-[var: Label]->(…)**

- *Q: Find the prices and bars to get the Sam Adams beer.*

SQL	Neo4j Cypher
FROM Sells s	**MATCH** (bar:Bar)-[s:Sells]->(beer:Beer)
WHERE s.beer = “Sam Adams”	**WHERE** beer.name = “Sam Adams”
SELECT s.price, s.bar	**RETURN** s.price, bar.name

Using Properties of Nodes and Relationships

- Node: **(var: Label {Properties})**
- Relationship: **(…)-[var: Label {Properties}]->(…)**

- *Q: Find the prices and bars to get the Sam Adams beer.*

```
MATCH (bar:Bar)-[sell:Sells]->(beer:Beer)
WHERE beer.name = "Sam Adams"
RETURN sell.price, bar.name
```

```
MATCH (bar:Bar)-[sell:Sells]->(beer:Beer {name:"Sam Adams"})
RETURN sell.price, bar.name
```

MATCH-WHERE-RETURN: Join Queries

- In SQL, you assemble the tables involved using joins.
- In a graph DB, you specify a pattern to traverse the graph.

- *Q: Find the beers brewed by AB InBev and sold at a price less than $5.*

SQL	Neo4j Cypher
FROM Beers b, Sells s	**MATCH** (bar:Bar)-[sell:Sells]->(beer:Beer {brewer : "AB InBev" })
WHERE b.name = s.beer AND b.brewer = "AB InBev" AND s.price < 5	**WHERE** sell.price < 5
SELECT b.name, s.price, s.bar	**RETURN** beer.name, sell.price, bar.name

Basic Query Examples

- *Q1: Find the beers with alcohol greater than 5%.*
- *Q2: Find the prices and bars to get the Sam Adams beer.*
- *Q3: Find the beers brewed by AB InBev and sold at a price less than $5.*

Food for Thought

Why does a graph database need “patterns”?
What does this notion correspond in SQL?

Querying Graph Databases: Advanced Capabilities

Querying Databases: The Non-relational Ways

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Kevin C.C. Chang, Professor

Computer Science @ Illinois

Learning Objectives

By the end of this video, you will be able to:

- Describe the aggregate framework in Neo4j Cypher.
- Explain how chaining of subqueries works.
- Write queries with these advanced capabilities.

Aggregate

```
SELECT beer.name, AVG(sell.price)
FROM Beers beer, Sells sell
WHERE beer.name = sell.beer
      and beer.brewer = "AB InBev"
GROUP BY beer.name
HAVING COUNT(sell.bar) >= 2
```

- Similar to SQL.
  - By applying aggregate functions over groups.
- No separate “GROUP BY” clause.
  - Simply put grouping attributes in “RETURN” together with aggregates.
  - RETURN $A_1, \ldots, A_n, F_1, \ldots, F_m$
    - Equivalent to:

      SELECT $A_1, \ldots, A_n, F_1, \ldots, F_m$

      GROUP BY $A_1, \ldots, A_n$
- No separate HAVING clause– use another “WHERE” by chaining.

Aggregate Query Examples

- *Q1: Find the highest price of beers on the market ~~and the most expensive beers with that price~~.*

- *Q2: Find the average price of each beer brewed by "AB InBev" ~~if it is sold at multiple bars~~.*

Chaining – Connecting Subqueries

- MATCH-WHERE-RETURN can be extended to a pipeline.
- MATCH-WHERE-(WITH-MATCH-WHERE)*-RETURN

- WITH clause is the connection from one stage to the next.
  - Specifies variables/functions to pass on, and give them aliases to refer to.
- E.g.,
  - … **WITH** beer.name **AS** beer, COUNT(*) **AS** countSales, AVG(s.price) **AS** avgPrice

Chained Query Examples

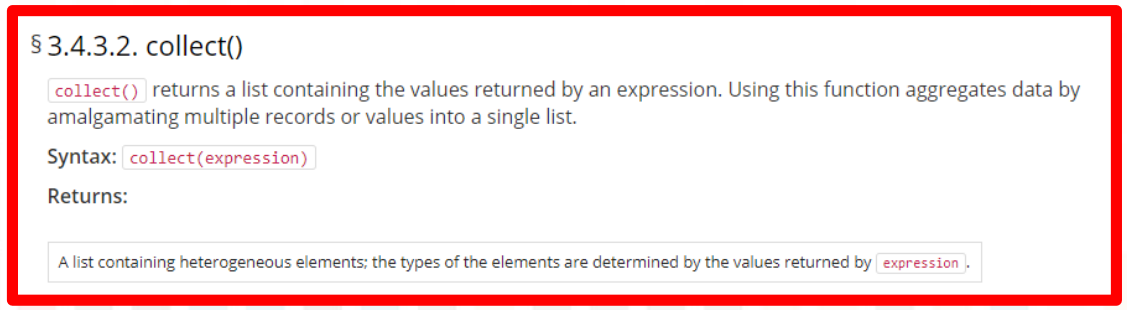
- *Q1: Find the highest price of beers on the market* ***and the most expensive beers with that price.***

- *Q2: Find the average price of each beer brewed by "AB InBev"* ***if it is sold at multiple bars.***

Food for Thought

In Neo4j Cypher, there is an interesting aggregate function COLLECT– which gathers elements of a group into a list. Can you imagine how it may be useful?

Hint: There is an UNWIND clause (like in MongoDB) to flatten a list, as the reverse of COLLECT, to access/use each element collected.



§ 3.4.3.2. collect()

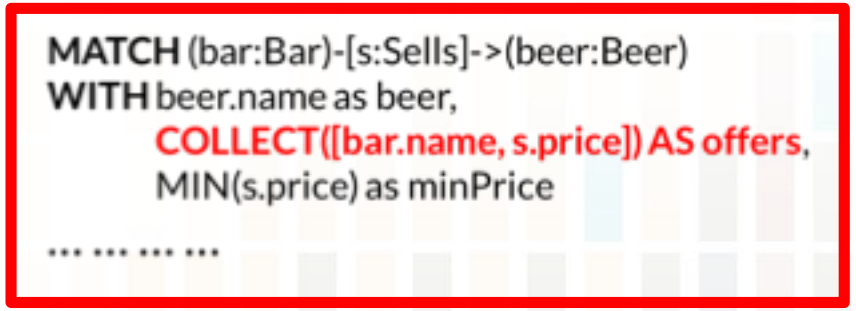
`collect()` returns a list containing the values returned by an expression. Using this function aggregates data by amalgamating multiple records or values into a single list.

Syntax: `collect(expression)`

Returns:

A list containing heterogeneous elements; the types of the elements are determined by the values returned by `expression`.

Cypher Manual 2017. Retrieved from https://neo4j.com/docs/developer-manual



```
MATCH (bar:Bar)-[s:Sells]->(beer:Beer)
WITH beer.name as beer,
     COLLECT([bar.name, s.price]) AS offers,
     MIN(s.price) as minPrice
... ... ... ...
```

Example query using COLLECT()

The End

Pretty Print JSON

- https://jsonformatter.org/json-pretty-print

Figure

```
db.CompleteBars.aggregate([
        {$unwind:"$sells"},
        {$match:{"sells.beer.brewer":"AB InBev"}},
        {$group:{_id:"$sells.beer.name",
           countSales:{$sum:1},
           avgPrice:{$avg:"$sells.price"}}},
        {$match:{"countSales":{$gt:1}}}
])
```