

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



ORACLE
BERKELEY DB



CouchDB
relax



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			

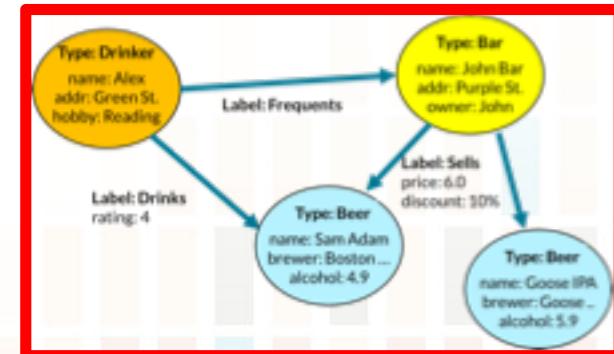
• • •

Contrasting non-relational models with relations

Now, do we assemble “documents”?



What do we assemble over a graph?



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



```
# MongoDB driver
import pymongo

conn=pymongo.MongoClient()
print "Connected to MongoDB."

db = conn.myDB

# Query Beers table
print "** Beers:"
for x in db.Beers.find():
    print x

# Query Bars table
print "** Bars:"
for x in db.Bars.find():
    print x
```

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.

Reference > mongo Shell Methods

mongo Shell Methods



On this page

• Collection	• Replication
• Cursor	• Sharding
• Database	• Subprocess
• Query Plan Cache	• Constructors
• Bulk Write Operation	• Connection
• User Management	• Native
• Role Management	

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
<code>db.collection.aggregate()</code>	Provides access to the aggregation pipeline.
<code>db.collection.count()</code>	Return a count of the number of documents in a collection or a view.
<code>db.collection.distinct()</code>	Returns an array of documents that have distinct values for the specified field.
<code>db.collection.find()</code>	Performs a query on a collection or a view and returns a cursor object.
<code>db.collection.findOne()</code>	Performs a query and returns a single document.
<code>db.collection.mapReduce()</code>	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.

Reference > Operators > Query and Projection Operators

Query and Projection Operators

On this page

- Query Selectors
- Projection Operators
- Additional Resources

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. FOR GIANT IDEAS

SOLUTIONS CLOUD CUSTOMERS R

Joins and Other Aggregation Enhancements Coming in MongoDB 3.2 (Part 1 of 3) – Introduction

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

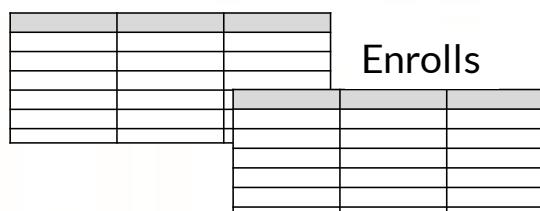
- Reduction: *Make a table smaller.*
 - Selection σ
 - Projection π
 - Combination: *Combine two tables.*
 - Set Union \cup
 - Set Difference $-$
 - Cartesian Product \times
 - Renaming ρ : *Change attribute name*

What is the major of Bugs Bunny?

Students

What courses are Bugs Bunny taking?

Students



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.

Basic Query Examples

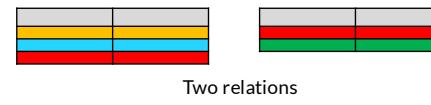
- *Q1: Find the beers that are brewed by AB InBev.*
- *Q2: Find the beers that are available at a price less than \$5.*

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

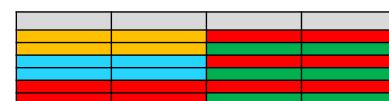
- Union: $R_1 \cup R_2$ (addition)
- Difference: $R_1 - R_2$ (like subtraction)



Union

- 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.

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 about their associated products, it must fetch the `products` 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.

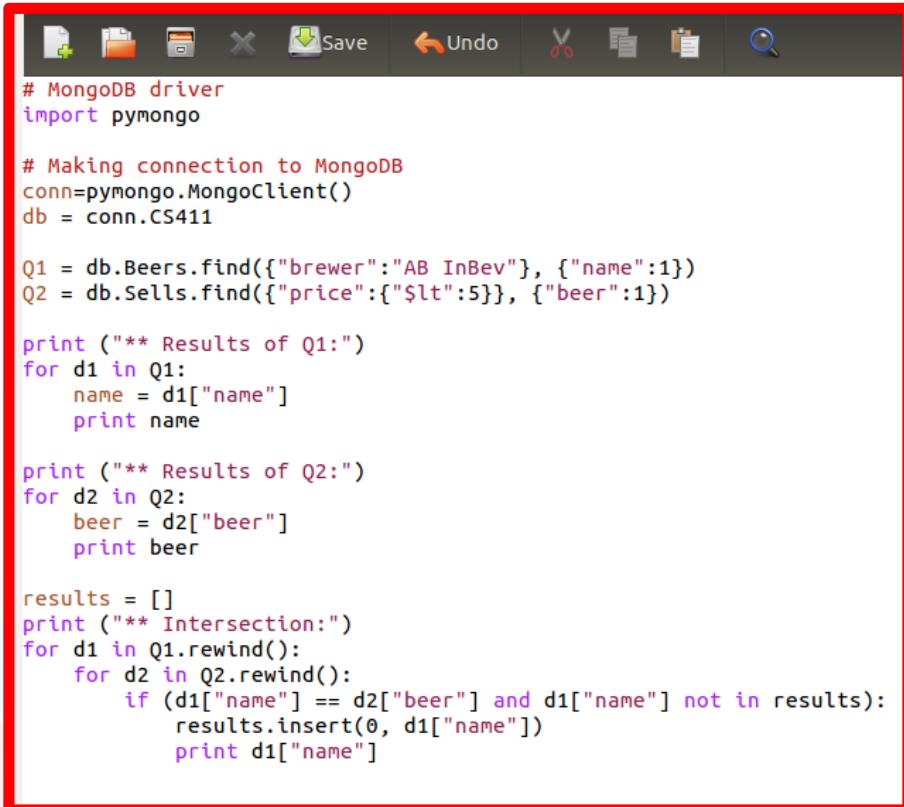
redacted, for now

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

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!

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



```
# MongoDB driver
import pymongo

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

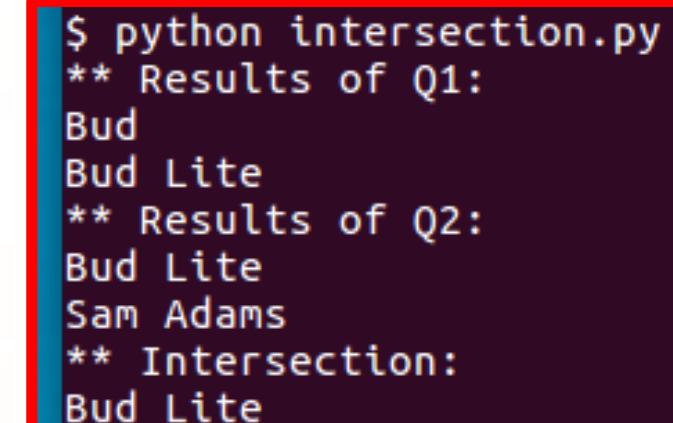
Q1 = db.Beers.find({"brewer":"AB InBev"}, {"name":1})
Q2 = db.Sells.find({"price":{$lt:5}}, {"beer":1})

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})

```
_id: ObjectId("59e665a09f1337a1ff41baac")
name: "Alex"
addr: "Green St"
hobby: "Reading"
frequents: Object
  name: "Sober Bar"
  addr: "Purple St"
  owner: "Jim"
drinks: Array
  0: Object
    beer: Object
      name: "Bud"
      brewer: "AB InBev"
      alcohol: 5
      rating: 3
  1: Object
    beer: Object
      name: "Sam Adams"
      brewer: "Boston Beer"
      alcohol: 4.9
      rating: 5
```

```
_id: ObjectId("59e665a09f1337a1ff41baad")
name: "Betty"
addr: "King St"
hobby: "Singing"
frequents: Object
  name: "Green Bar"
  addr: "Green St"
  owner: "Sally"
drinks: Array
  0: Object
    beer: Object
      name: "Sam Adams"
      brewer: "Boston Beer"
```

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

```
_id: ObjectId("59e665a09f1337a1ff41baac")
name: "Alex"
addr: "Green St"
hobby: "Reading"
✓frequents: Object
  name: "Sober Bar"
  addr: "Purple St"
  owner: "Jim"
✓drinks: Array
  ✓0: Object
    ✓beer: Object
      name: "Bud"
      brewer: "AB InBev"
      alcohol: 5
      rating: 3
    ✓1: Object
      ✓beer: Object
        name: "Sam Adams"
        brewer: "Boston Beer"
        alcohol: 4.9
        rating: 5

_id: ObjectId("59e665a09f1337a1ff41baad")
name: "Betty"
addr: "King St"
hobby: "Singing"
✓frequents: Object
  name: "Green Bar"
  addr: "Green St"
  owner: "Sally"
✓drinks: Array
  ✓0: Object
    ✓beer: Object
      name: "Sam Adams"
      brewer: "Boston Beer"
```

CompleteDrinkers

Embedding

```
_id: ObjectId("59e674dc51ff64a521081e90")
name: "Alex"
addr: "Green St"
hobby: "Reading"
frequents: ObjectId("59e62a3a93b695daed6adf33")
✓drinks: Array
  ✓0: Object
    beer: ObjectId("59e62a3a93b695daed6adf33")
    rating: 3
  ✓1: Object
    beer: ObjectId("59e3a4889108219c38e8a95b")
    rating: 5

_id: ObjectId("59e674dc51ff64a521081e91")
name: "Betty"
addr: "King St"
hobby: "Singing"
frequents: ObjectId("59e62a3a93b695daed6adf34")
✓drinks: Array
  ✓0: Object
    beer: ObjectId("59e3a4889108219c38e8a95b")
    rating: 2
```

CompleteRefDrinkers

Referencing

Querying into Referenced Documents

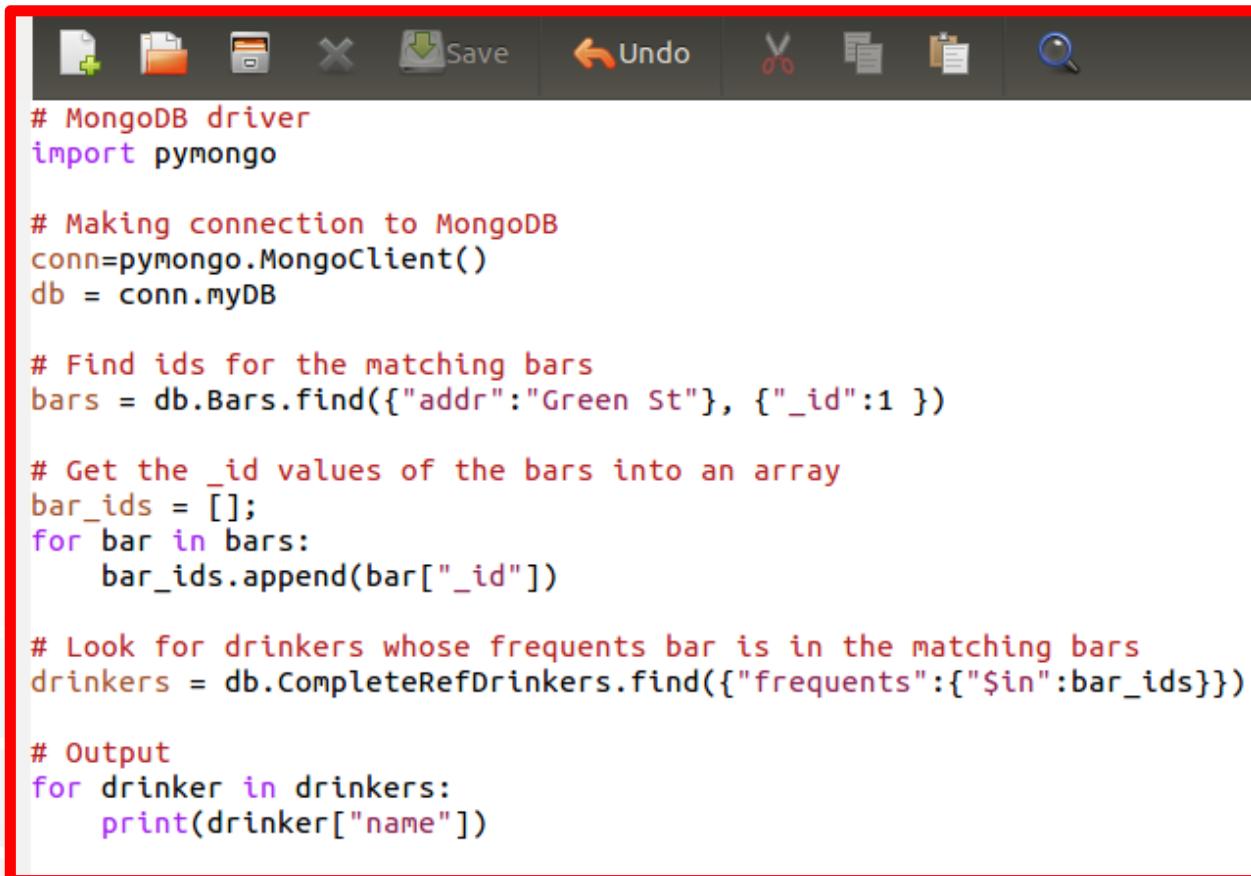
- *Q: Drinkers who frequent a bar at address Y?*
- db.CompleteDrinkers.find({"frequents.addr":"Green St"})?

```
_id: ObjectId("59e674dc51ff64a521081e90")
name: "Alex"
addr: "Green St"
hobby: "Reading"
frequents: ObjectId("59e62a3a93b695daed6adf33")
drinks: Array
  0: Object
    beer: ObjectId("59e62a3a93b695daed6adf33")
    rating: 3
  1: Object
    beer: ObjectId("59e3a4889108219c38e8a95b")
    rating: 5

_id: ObjectId("59e674dc51ff64a521081e91")
name: "Betty"
addr: "King St"
hobby: "Singing"
frequents: ObjectId("59e62a3a93b695daed6adf34")
drinks: Array
  0: Object
    beer: ObjectId("59e3a4889108219c38e8a95b")
    rating: 2
```

Querying into Referenced Documents – Doing It Yourself!

- Retrieve referenced objects. Check if match conditions.



The screenshot shows a code editor window with a red border. The toolbar at the top includes icons for file operations (New, Open, Save, Undo, Redo, Cut, Copy, Paste, Find, and Search). Below the toolbar is a code editor area containing Python code for querying a MongoDB database using the pymongo driver. The code is as follows:

```
# MongoDB driver
import pymongo

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

# Find ids for the matching bars
bars = db.Bars.find({"addr":"Green St"}, {"_id":1 })

# Get the _id values of the bars into an array
bar_ids = []
for bar in bars:
    bar_ids.append(bar["_id"])

# Look for drinkers whose frequents bar is in the matching bars
drinkers = db.CompleteRefDrinkers.find({"frequents":{"$in":bar_ids}})

# Output
for drinker in drinkers:
    print(drinker["name"])
```

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).*

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.
options	document	<p><i>Changed in version 2.6:</i> 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.</p> <p>Optional. Additional options that <code>aggregate()</code> passes to the aggregate command.</p>

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.*

Aggregate Query Examples

- Q1: Find the average grade of CS411. Compare with CS423.
- Q2: : Find average prices 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
```

SQL Aggregate: The Overall Framework

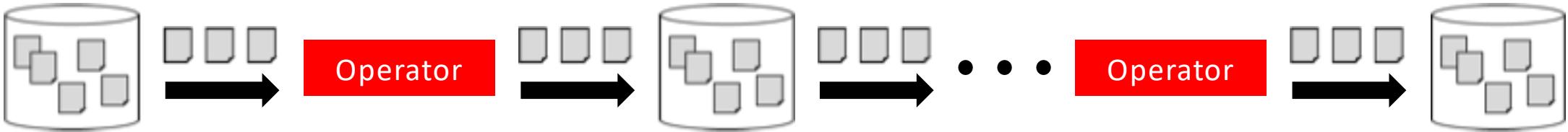
name	brewer	alcohol
Sam Adams	Boston Beer	4.9
Bud	All InBev	5
Bud Lite	All InBev	4.2
Coors	Coors	5

bar	beer	price
Sober Bar	Bud	5
Sober Bar	Bud Lite	3
Sober Bar	Sam Adams	6
Green Bar	Sam Adams	4.5
Green Bar	Bud	2
Green Bar	Coors	
Purple Bar	Sam Adams	5



Aggregate query framework

The Pipeline Framework



```
1<| [ {  
2+   "name": "Sober Bar",  
3+   "addr": "Purple St",  
4+   "owner": "Jim",  
5+   "sells": [  
6+     {  
7+       "beer": {  
8+         "name": "Bud",  
9+         "brewer": "AB InBev",  
10+        "alcohol": 5  
11+      },  
12+      "price": 5,  
13+      "discount": 0.05  
14+    },  
15+    {  
16+      "beer": {  
17+        "name": "Bud Lite",  
18+        "brewer": "AB InBev",  
19+        "alcohol": 4.2  
20+      },  
21+      "price": 3,  
22+      "discount": 0  
23+    },  
24+    {  
25+      "beer": {  
26+        "name": "Sam Adams",  
27+        "brewer": "Boston Beer",  
28+        "alcohol": 4.9  
29+      },  
30+      "price": 6,  
31+      "discount": 0.1  
32+    }  
33+  ]  
34+ },  
35+ {  
36+   "name": "Green Bar",  
37+   "addr": "Green St",  
38+   "owner": "Sally",  
39+   "sells": [  
40+     {  
41+       "beer": {  
42+         "name": "Sam Adams",  
43+         "brewer": "Boston Beer",  
44+         "alcohol": 4.9  
45+       },  
46+       "price": 4.5,  
47+       "discount": 0.2  
48+     }  
49+   ]  
50+ }]
```

\$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}}}  
])
```

\$match

```
1<| [ {  
2+   "_id": "Bud",  
3+   "countSales": 3,  
4+   "avgPrice": 3.5  
5+ }  
6+ ]  
7+ ]  
8+ ]
```

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}}}
])
```

```
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 } ] } ]
```



```
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": "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": { "beer": { "name": "Sam Adams", "brewer": "Boston Beer", "alcohol": 4.9 }, "price": 4.5, "discount": 0.2 } }, { "name": "Green Bar", "addr": "Green St", "owner": "Sally", "sells": { "beer": { "name": "Sam Adams", "brewer": "Boston Beer", "alcohol": 4.9 }, "price": 6, "discount": 0.1 } } ]
```

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}}}
])
```

A screenshot of the MongoDB Compass interface showing the 'CompleteBars' collection. It contains four documents, each representing a bar with its address, owner, and a 'sells' array. The 'sells' array contains beer details like name, brewer, alcohol percentage, price, and discount. The first document is highlighted with a red box.

\$unwind

A screenshot of the MongoDB Compass interface showing the state after the '\$unwind' stage. The original four documents have been expanded into 12 individual beer documents. Each document includes the bar's name, address, and owner, followed by the beer's name, brewer, alcohol percentage, price, and discount. The first beer document from the first bar is highlighted with a red box.

\$match

A screenshot of the MongoDB Compass interface showing the final state of the aggregate pipeline. Only two documents remain, corresponding to the 'Sober Bar' and 'Green Bar'. Both documents contain the bar's information and a single beer document from the '\$unwind' stage. The first beer document from the 'Sober Bar' is highlighted with a red box.

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}}}
])
```

```
1: {
  "name": "Sober Bar",
  "addr": "Purple St",
  "owner": "Jim",
  "sells": [
    {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 3,
      "discount": 0.05
    },
    {
      "beer": {
        "name": "Bud Lite",
        "brewer": "AB InBev",
        "alcohol": 4.2
      },
      "price": 3,
      "discount": 0
    },
    {
      "beer": {
        "name": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "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": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "name": "Green Bar",
  "addr": "Green St",
  "owner": "Sally",
  "sells": [
    {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 2,
      "discount": 0
    },
    {
      "beer": {
        "name": "Bud Lite",
        "brewer": "AB InBev",
        "alcohol": 4.2
      },
      "price": 3,
      "discount": 0
    },
    {
      "beer": {
        "name": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "name": "Purple Bar",
  "addr": "Purple St",
  "owner": "Paul",
  "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": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}
```

\$unwind

```
1: {
  "name": "Sober Bar",
  "addr": "Purple St",
  "owner": "Jim",
  "sells": [
    {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 3,
      "discount": 0.05
    },
    {
      "beer": {
        "name": "Bud Lite",
        "brewer": "AB InBev",
        "alcohol": 4.2
      },
      "price": 3,
      "discount": 0
    },
    {
      "beer": {
        "name": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "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": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "name": "Green Bar",
  "addr": "Green St",
  "owner": "Sally",
  "sells": [
    {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 2,
      "discount": 0
    },
    {
      "beer": {
        "name": "Bud Lite",
        "brewer": "AB InBev",
        "alcohol": 4.2
      },
      "price": 3,
      "discount": 0
    },
    {
      "beer": {
        "name": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "name": "Purple Bar",
  "addr": "Purple St",
  "owner": "Paul",
  "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": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}
```

\$match

```
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": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "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": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "name": "Green Bar",
  "addr": "Green St",
  "owner": "Sally",
  "sells": [
    {
      "beer": {
        "name": "Bud",
        "brewer": "AB InBev",
        "alcohol": 5
      },
      "price": 2,
      "discount": 0
    },
    {
      "beer": {
        "name": "Bud Lite",
        "brewer": "AB InBev",
        "alcohol": 4.2
      },
      "price": 3,
      "discount": 0
    },
    {
      "beer": {
        "name": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}, {
  "name": "Purple Bar",
  "addr": "Purple St",
  "owner": "Paul",
  "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": "San Adams",
        "brewer": "Boston Beer",
        "alcohol": 4.8
      },
      "price": 4,
      "discount": 0.2
    }
  ]
}
```

\$group

```
1: [
  2:   {
  3:     "_id": "Bud Lite",
  4:     "countSales": 1,
  5:     "avgPrice": 3
  6:   },
  7:   {
  8:     "_id": "Bud",
  9:     "countSales": 3,
 10:    "avgPrice": 3.5
 11:  }
 12]
```

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}}}
])
```

```
1: [
2:   {
3:     "sells": [
4:       {
5:         "beer": {
6:           "name": "Budweiser",
7:           "brewer": "Budweiser"
8:         }
9:       }
10:      ],
11:      "countSales": 1,
12:      "avgPrice": 3.0
13:      "discount": 0.05
14:    },
15:    {
16:      "sells": [
17:        {
18:          "beer": {
19:              "name": "Bud Lite",
20:              "brewer": "AB InBev"
21:            }
22:          }
23:        ],
24:        "countSales": 3,
25:        "avgPrice": 3.5
26:        "discount": 0.0
27:      }
28:    ],
29:    "countSales": 4,
30:    "avgPrice": 3.25
31:    "discount": 0.025
32:  },
33:  {
34:    "sells": [
35:      {
36:        "beer": {
37:            "name": "Budweiser",
38:            "brewer": "Budweiser"
39:          }
40:        }
41:      ],
42:      "countSales": 1,
43:      "avgPrice": 3.0
44:      "discount": 0.05
45:    },
46:    {
47:      "sells": [
48:        {
49:          "beer": {
50:              "name": "Budweiser",
51:              "brewer": "Budweiser"
52:            }
53:          }
54:        ],
55:        "countSales": 1,
56:        "avgPrice": 3.0
57:        "discount": 0.05
58:      }
59:    ],
60:    "countSales": 2,
61:    "avgPrice": 3.0
62:    "discount": 0.025
63:  },
64:  {
65:    "sells": [
66:      {
67:        "beer": {
68:            "name": "Budweiser",
69:            "brewer": "Budweiser"
70:          }
71:        }
72:      ],
73:      "countSales": 1,
74:      "avgPrice": 3.0
75:      "discount": 0.05
76:    },
77:    {
78:      "sells": [
79:        {
80:          "beer": {
81:              "name": "Budweiser",
82:              "brewer": "Budweiser"
83:            }
84:          }
85:        ],
86:        "countSales": 1,
87:        "avgPrice": 3.0
88:        "discount": 0.05
89:      }
90:    ],
91:    "countSales": 2,
92:    "avgPrice": 3.0
93:    "discount": 0.025
94:  }
95: ]
```

\$unwind

```
1: [
2:   {
3:     "sells": [
4:       {
5:         "beer": {
6:             "name": "Budweiser",
7:             "brewer": "Budweiser"
8:           }
9:         }
10:        ],
11:        "countSales": 1,
12:        "avgPrice": 3.0
13:        "discount": 0.05
14:      },
15:      {
16:        "sells": [
17:          {
18:            "beer": {
19:                "name": "Bud Lite",
20:                "brewer": "AB InBev"
21:              }
22:            }
23:          ],
24:          "countSales": 3,
25:          "avgPrice": 3.5
26:          "discount": 0.0
27:        }
28:      ],
29:      "countSales": 4,
30:      "avgPrice": 3.25
31:      "discount": 0.025
32:    },
33:    {
34:      "sells": [
35:        {
36:          "beer": {
37:              "name": "Budweiser",
38:              "brewer": "Budweiser"
39:            }
40:          }
41:        ],
42:        "countSales": 1,
43:        "avgPrice": 3.0
44:        "discount": 0.05
45:      },
46:      {
47:        "sells": [
48:          {
49:            "beer": {
50:              "name": "Budweiser",
51:              "brewer": "Budweiser"
52:                }
53:              }
54:            ],
55:            "countSales": 1,
56:            "avgPrice": 3.0
57:            "discount": 0.05
58:          }
59:        ],
60:        "countSales": 2,
61:        "avgPrice": 3.0
62:        "discount": 0.025
63:      },
64:      {
65:        "sells": [
66:          {
67:            "beer": {
68:                "name": "Budweiser",
69:                "brewer": "Budweiser"
70:                  }
71:                  }
72:                  ],
73:                  "countSales": 1,
74:                  "avgPrice": 3.0
75:                  "discount": 0.05
76:                },
77:                {
78:                  "sells": [
79:                    {
80:                      "beer": {
81:                          "name": "Budweiser",
82:                          "brewer": "Budweiser"
83:                            }
84:                            }
85:                            ],
86:                            "countSales": 1,
87:                            "avgPrice": 3.0
88:                            "discount": 0.05
89:                          }
90:                        ],
91:                        "countSales": 2,
92:                        "avgPrice": 3.0
93:                        "discount": 0.025
94:                      }
95:                    ]
```

\$match

```
1: [
2:   {
3:     "sells": [
4:       {
5:         "beer": {
6:             "name": "Budweiser",
7:             "brewer": "Budweiser"
8:           }
9:         }
10:        ],
11:        "countSales": 1,
12:        "avgPrice": 3.0
13:        "discount": 0.05
14:      },
15:      {
16:        "sells": [
17:          {
18:            "beer": {
19:                "name": "Bud Lite",
20:                "brewer": "AB InBev"
21:                  }
22:                  }
23:                ],
24:                "countSales": 3,
25:                "avgPrice": 3.5
26:                "discount": 0.0
27:              }
28:            ],
29:            "countSales": 4,
30:            "avgPrice": 3.25
31:            "discount": 0.025
32:          },
33:          {
34:            "sells": [
35:              {
36:                "beer": {
37:                    "name": "Budweiser",
38:                    "brewer": "Budweiser"
39:                      }
40:                      }
41:                      ],
42:                      "countSales": 1,
43:                      "avgPrice": 3.0
44:                      "discount": 0.05
45:                    },
46:                    {
47:                      "sells": [
48:                        {
49:                          "beer": {
50:                              "name": "Budweiser",
51:                              "brewer": "Budweiser"
52:                                }
53:                                }
54:                                ],
55:                                "countSales": 1,
56:                                "avgPrice": 3.0
57:                                "discount": 0.05
58:                              }
59:                            ],
60:                            "countSales": 2,
61:                            "avgPrice": 3.0
62:                            "discount": 0.025
63:                          },
64:                          {
65:                            "sells": [
66:                              {
67:                                "beer": {
68:                                    "name": "Budweiser",
69:                                    "brewer": "Budweiser"
70:                                      }
71:                                      }
72:                                      ],
73:                                      "countSales": 1,
74:                                      "avgPrice": 3.0
75:                                      "discount": 0.05
76:                                    },
77:                                    {
78:                                      "sells": [
79:                                        {
80:                                          "beer": {
81:                                              "name": "Budweiser",
82:                                              "brewer": "Budweiser"
83:                                                }
84:                                                }
85:                                                ],
86:                                                "countSales": 1,
87:                                                "avgPrice": 3.0
88:                                                "discount": 0.05
89:                                              }
90:                                            ],
91:                                            "countSales": 2,
92:                                            "avgPrice": 3.0
93:                                            "discount": 0.025
94:                                          }
95:                                        ]
```

\$group

```
1: [
2:   {
3:     "_id": "Bud Lite",
4:     "countSales": 1,
5:     "avgPrice": 3
6:   },
7:   {
8:     "_id": "Bud",
9:     "countSales": 3,
10:    "avgPrice": 3.5
11:  }
12: ]
```

\$match

```
1: [
2:   {
3:     "_id": "Bud",
4:     "countSales": 3,
5:     "avgPrice": 3.5
6:   }
7: ]
```

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

```
_id: ObjectId("59e665a09f1337a1ff41baac")
name: "Alex"
addr: "Green St"
hobby: "Reading"
✓ frequents: Object
  name: "Sober Bar"
  addr: "Purple St"
  owner: "Jim"
✓ drinks: Array
  ✓ 0: Object
    ✓ beer: Object
      name: "Bud"
      brewer: "AB InBev"
      alcohol: 5
      rating: 3
    ✓ 1: Object
      ✓ beer: Object
        name: "Sam Adams"
        brewer: "Boston Beer"
        alcohol: 4.9
        rating: 5

_id: ObjectId("59e665a09f1337a1ff41baad")
name: "Betty"
addr: "King St"
hobby: "Singing"
✓ frequents: Object
  name: "Green Bar"
  addr: "Green St"
  owner: "Sally"
✓ drinks: Array
  ✓ 0: Object
    ✓ beer: Object
      name: "Sam Adams"
      brewer: "Boston Beer"

_id: ObjectId("59e6294e93b695daed6adf27")
name: "Sober Bar"
addr: "Purple St"
owner: "Jim"
✓ sells: Array
  ✓ 0: Object
    ✓ beer: Object
      name: "Bud"
      brewer: "AB InBev"
      alcohol: 5
      price: 5
      discount: 0.05
    ✓ 1: Object
      ✓ beer: Object
        name: "Bud Lite"
        brewer: "AB InBev"
        alcohol: 4.2
        price: 3
        discount: 0
    ✓ 2: Object
      ✓ beer: Object
        name: "Sam Adams"
        brewer: "Boston Beer"
        alcohol: 4.9
        price: 6
        discount: 0.1
```

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).

Why Not Joins or Other Binary Ops?

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's possible that the data you need spans across 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 sales collection. The application can then use the `_id` reference to calculate documents from the products collection. Prior to MongoDB 3.2, this work is implemented in application code. However, this will automatically be applied to the application and requires no code from the database, which can dramatically reduce latency.

Database References, 2017. Retrieved from <https://www.mongodb.com/developer/tutorials/database-references/>

```
..._id: ObjectId("59e665a99f1337a1ff41baad")
  name: "Alex"
  addr: "Green St"
  hobby: "Reading"
  - Frequent: Object
    name: "sober Bar"
    addr: "Purple St"
    owner: "Jim"
  - drinks: Array
    - 0: Object
      - beer: Object
        name: "Bud"
        brewer: "AB InBev"
        alcohol: 5
        rating: 3
    - 1: Object
      - beer: Object
        name: "Sam Adams"
        brewer: "Boston Beer"
        alcohol: 4.9
        rating: 5

..._id: ObjectId("59e6294e93b695daed6adfd21")
  name: "sober Bar"
  addr: "Purple St"
  owner: "Jim"
  - sells: Array
    - 0: Object
      - beer: Object
        name: "Bud"
        brewer: "AB InBev"
        alcohol: 5
        price: 5
        discount: 0.05
    - 1: Object
      - beer: Object
        name: "Bud Lite"
        brewer: "AB InBev"
        alcohol: 4.2
        price: 3
        discount: 0
    - 2: Object
      - beer: Object
        name: "Sam Adams"
        brewer: "Boston Beer"
        alcohol: 4.9
        price: 6
        discount: 0.1

..._id: ObjectId("59e665a99f1337a1ff41baad")
  name: "Betty"
  addr: "King St"
  hobby: "Singing"
  - frequents: Object
    name: "Green Bar"
    addr: "Green St"
    owner: "Sally"
  - drinks: Array
    - 0: Object
      - beer: Object
        name: "Sam Adams"
        brewer: "Boston Beer"
```

Really? The Chase for Joins Started Early On!

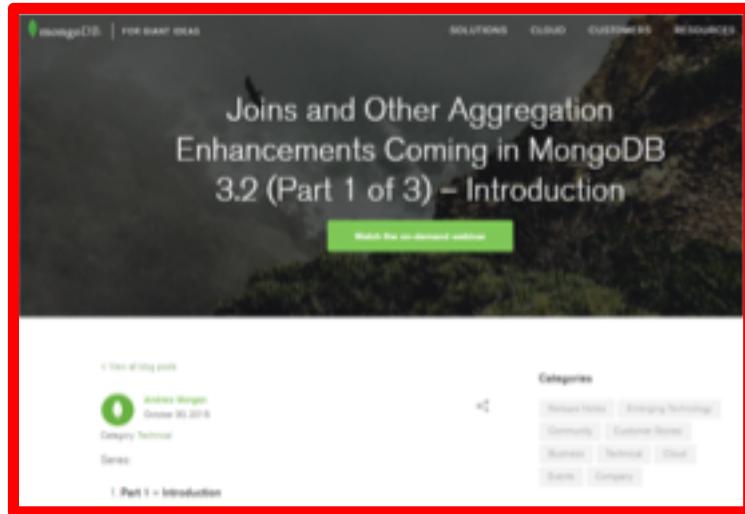
A screenshot of a Stack Overflow search results page for the query "mongodb join". The results show three posts:

- Q: How do I perform the SQL Join equivalent in MongoDB? [duplicate]**
How do I perform the SQL Join equivalent in MongoDB? For example say you have two collections (users and comments) and I want to pull all the comments with pid=444 along with the user info for each ...
asked Feb 28 '10 by [The Unknown](#)
343 votes
17 answers
- A: How to list all collections in the mongo shell?**
, you can: \$ mongo prodmongo/app --eval "db.getCollectionNames().join("\n")" MongoDB shell version: 3.2.10 connecting to: prodmongo/app Profiles Unit_Info ... "show collections" MongoDB shell version: 3.2.10 connecting to: prodmongo/app 2016-10-26T19:34:34.886-0400 E QUERY [thread1] SyntaxError: missing ; before statement @(shell eval):1:5 \$ mongo ...
answered Jan 14 '12 by [AdaTheDev](#)
932 votes
- Q: MongoDB and "joins" [duplicate]**
I'm sure MongoDB doesn't officially support "joins". What does this mean? Does this mean "We cannot connect two collections(tables) together."? I think if we put the value for _id in collection A ... to the other_id in collection B, can we simply connect two collections? If my understanding is correct, MongoDB can connect two tables together, say, when we run a query. This is done by "Reference ..."
asked Nov 1 '10 by [TK](#)
151 votes
11 answers

A screenshot of the MongoDB entry on Wikipedia. The page provides the following information:

Developer(s)	MongoDB Inc.
Initial release	February 2009, 11 ^[1]
Stable release	3.4.9 ^[2] / 11 September 2017; 37 days ago
Preview release	3.5.12 ^[3] / 22 August 2017; 57 days ago
Repository	github.com/mongodb/mongo ^[4]
Development status	Active
Written in	C++, C and JavaScript
Operating system	Windows Vista and later, Linux, OS X 10.7 and later, Solaris, ^[4] FreeBSD ^[5]
Available in	English
Type	Document-oriented database
License	Various; see § Licensing
Website	www.mongodb.com ^[4]

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>

The Most Awaited Feature "SQL Joins" is Now Available in MongoDB 3.2

MONGODB, TECHNOLOGY

17 NOV 2016 BY DIBBALOGOAL 0 COMMENTS

Share this Blog



SQL Joins are used to combine documents/rows from 2 or more tables based upon common field present in them.

MongoDB 3.2 launched the most awaited feature "joins" which is supported in SQL database however was not present in the earlier version of MongoDB. This feature will change the way you design your database schema and application using MongoDB.

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>

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

Outer Equi-Join: Collection-Centric Perspective

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

Aside - What is a Left Outer Equi-Join?

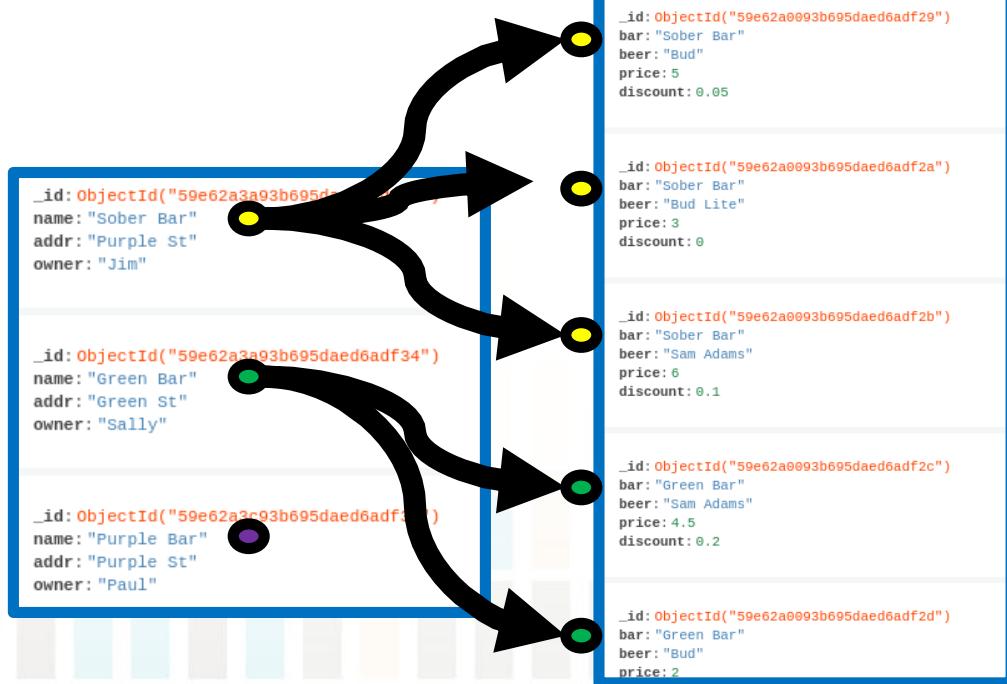
A left outer equi-join produces a result set that contains data for all documents from the left table (collection) together with data from the right table (collection) for documents where there is a match with documents from the left table (collection). This is illustrated in Figure 2.



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



```
1 [ {
2   {
3     "name": "Sober Bar",
4     "addr": "Purple St",
5     "owner": "Jim",
6     "menu": [
7       {
8         "beer": "Bud",
9         "price": 5,
10        "discount": 0.05
11      },
12      {
13        "beer": "Bud Lite",
14        "price": 3,
15        "discount": 0
16      },
17      {
18        "beer": "Sam Adams",
19        "price": 6,
20        "discount": 0.1
21      }
22   },
23   {
24     "name": "Green Bar",
25     "addr": "Green St",
26     "owner": "Sally",
27     "menu": [
28       {
29         "beer": "Sam Adams",
30         "price": 4.5,
31         "discount": 0.2
32       },
33       {
34         "beer": "Coors",
35         "price": null,
36         "discount": null
37       }
38     ]
39   }
40 }
```

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.*

*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"}}
])
```



```
1: [
2:   { "realBuddy1": "Cindy",
3:     "realBuddy2": "Alex"
4:   }
5: ]
```

*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.

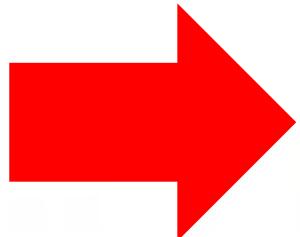
FridayNight Database— Now It's a Graph!

Drinkers				
name	addr	hobby	bar	beer
Bars				
name	addr	owner	beer	
Breweries				
name	brewer	alcohol		
Frequents				
drinker			bar	

Relational database

```
[{"id": "0000001000",  
 "name": "Samuel Adams",  
 "brewery": [  
     {"name": "Boston Beer Company",  
      "location": "Boston, Massachusetts"  
    },  
    {"alcohol": 4.5,  
     "type": "Lager",  
     "year introduced": 1984,  
     "variants": [  
         {"alcohol": 4.5,  
          "type": "Lager",  
          "name": "Samuel Adams Light",  
          "brewery": [  
              {"name": "Boston Beer Company",  
               "location": "Boston, Massachusetts"  
            },  
            {"alcohol": 5.1,  
             "type": "Lager",  
             "year introduced": 1993  
           }  
        ]  
    ]  
}
```

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.

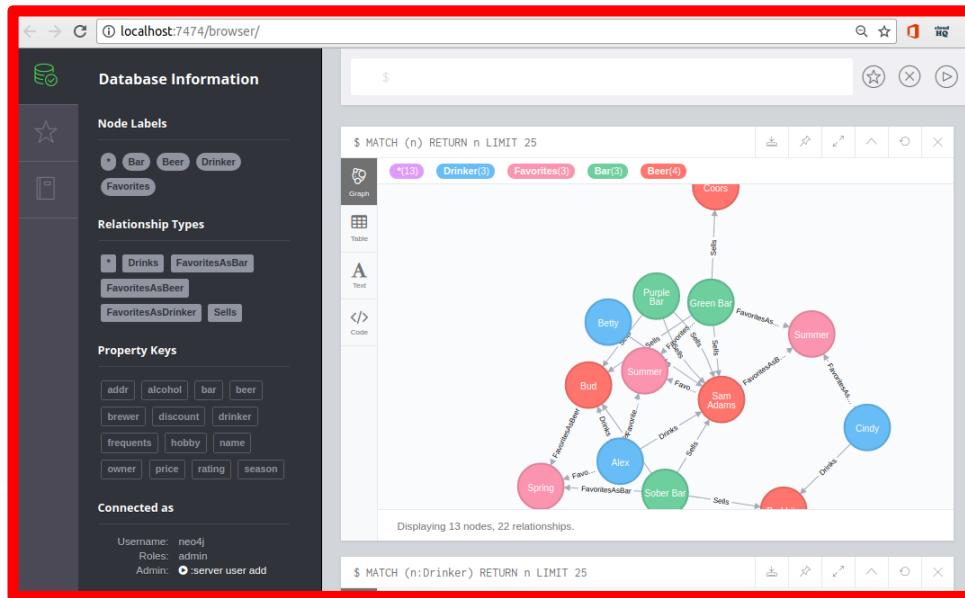
Cypher is designed to be a humane query language, suitable for both developers and (importantly, we think) operations professionals. Our guiding goal is to make the simple things easy, and the complex things possible. Its constructs are based on English prose and neat iconography which helps to make 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.

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

A screenshot of a Python code editor with a red border around the code area. The code uses the neo4jrestclient library to connect to a Neo4j database at http://localhost:7474. It defines a query q to find beers favored by a drinker named Alex. The results are stored in results, and each result's name is printed. The code includes imports for client and GraphDatabase, a connection setup, the query definition, and a loop to print the results.

```
from neo4jrestclient import client
from neo4jrestclient.client import GraphDatabase

# Connect to DB
db = GraphDatabase("http://localhost:7474",
                   username="neo4j", password="i-love-tea")

q = 'MATCH (d:Drinker)-[:Drinks]->(b:Beer)' \
    + 'WHERE d.name = "Alex" RETURN b'

# Execute query
results = db.query(q, returns=(client.Node))

for r in results:
    print(r[0]["name"])
```

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.*

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

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

```
SELECT A1, ..., An, F1, ..., Fm
GROUP BY A1, ..., An
```
- No separate HAVING clause— use another “WHERE” by chaining.

```
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
```

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.***

*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 `expression`.

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

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}}}
])
```