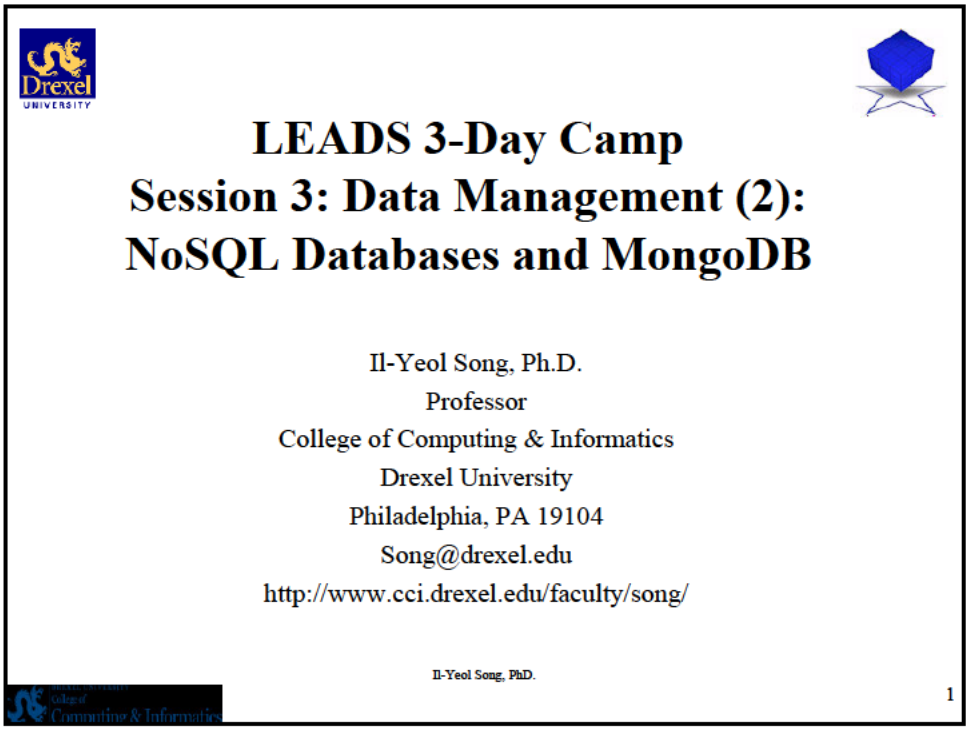
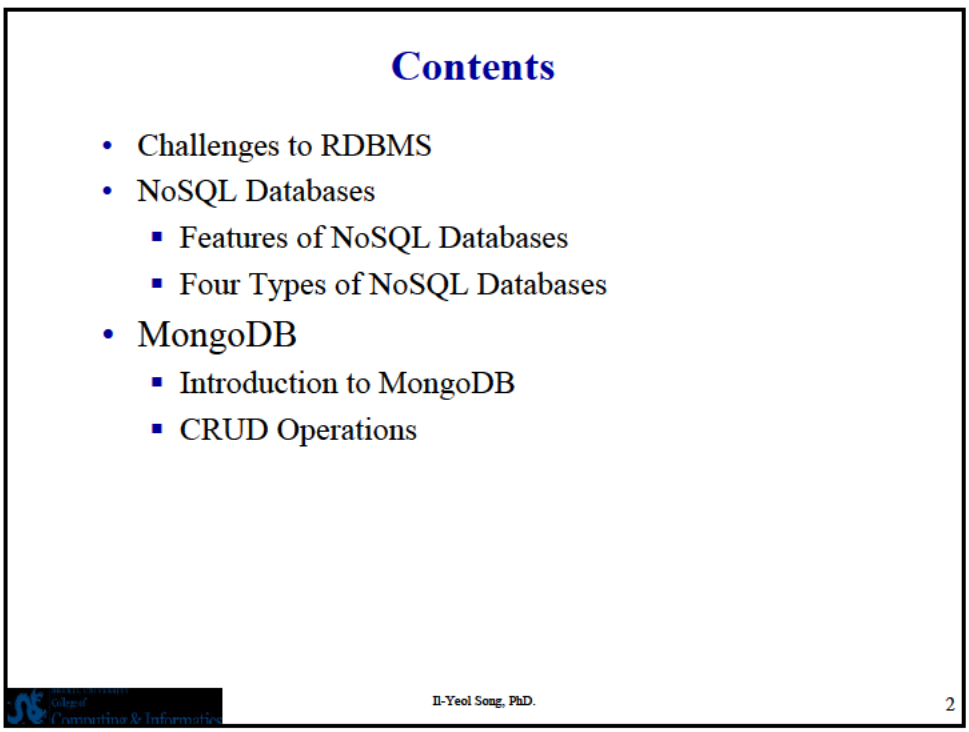
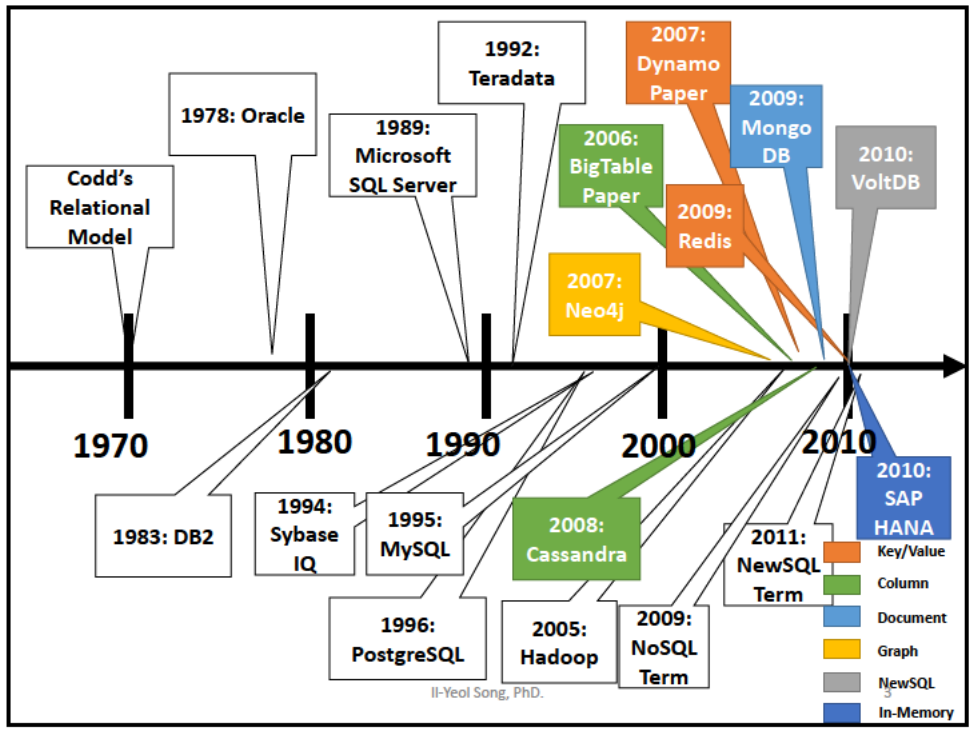

Drexel
UNIVERSITY
LEADS 3-Day Camp
Session 3: Data Management (2):
NoSQL Databases and MongoDB
Il-Yeol Song, Ph.D.
Professor
College of Computing & Informatics
Drexel University
Philadelphia, PA 19104
Song@drexel.edu
http://www.cci.drexel.edu/faculty/song/
Il-Yeol Song, PhD.
1

1


Contents
Challenges to RDBMS
NoSQL Databases
Features of NoSQL Databases
Four Types of NoSQL Databases
MongoDB
Introduction to MongoDB
CRUD Operations
Il-Yeol Song, PhD.
2

2



3

Characteristics of RDBMS

- Record-oriented persistent storage in table forms
- Structured data predefined by a schema
- Powerful standard query language--SQL
- Transactions with ACID properties
  - A group of statements executed together
  - All or nothing
  - Ex: Transfer $100 from Saving Account A to Checking Account B
  - Reliable even in a distributed databases

4

Il-Yeol Song, PhD.

4

Challenges to RDBs

- RDBMSs were not designed to be distributed (joins, RIs)
- Enforcing ACID is expensive (locking, background processes, disk access)
- Explosion of data (Volume)
- New applications and data sources: e-commerce, social networks, IOTs, cloud
  - Most big data are unstructured: RDB Schema causes a problem
  - Different types of data (Variety)
- Specialized applications: IOT, sensors, real-time applications: (Velocity)
- Globally distributed systems
  - Replication, Scalability

=> Not good for **big data**, **real-time**, and **cloud**

Features of NoSQL Databases (1)

- Manage ***non-relational* data**: Semi and non-structured data
- Stores ***aggregated objects***:
  - Stores all the related data together (denormalzied objects)
    - No complex relationships
  - Could cause redundant data and update anomaly
  - Write-once-read-many applications
- ***Schema-on-read, mostly***
  - Some column stores need to define the schema in advance, but columns can be added dynamically
- Built on ***the scale-out architecture***
  - Scalability, fault tolerance, High availability (HA)
  - Support ***automatic sharding***
    - Data are automatically distributed to nodes based on some fields

Major Features of NoSQL Databases (2)

- Soften the ACID property:
  - **BASE**: *Basically **Available**, Soft state, **Eventual consistency***
    - ***NoSQLs focuse on scalability and availability***
  - The data (and its replicas) will become consistent at some point in time after each transaction
  - Supports high throughput and low latency
- Open-sourced
- Four types:
  - Key-value stores, Wide Column stores, Document stores
  - Graph databases
- Designed for **real-time OLTP system for non-uniform, big data**.

7

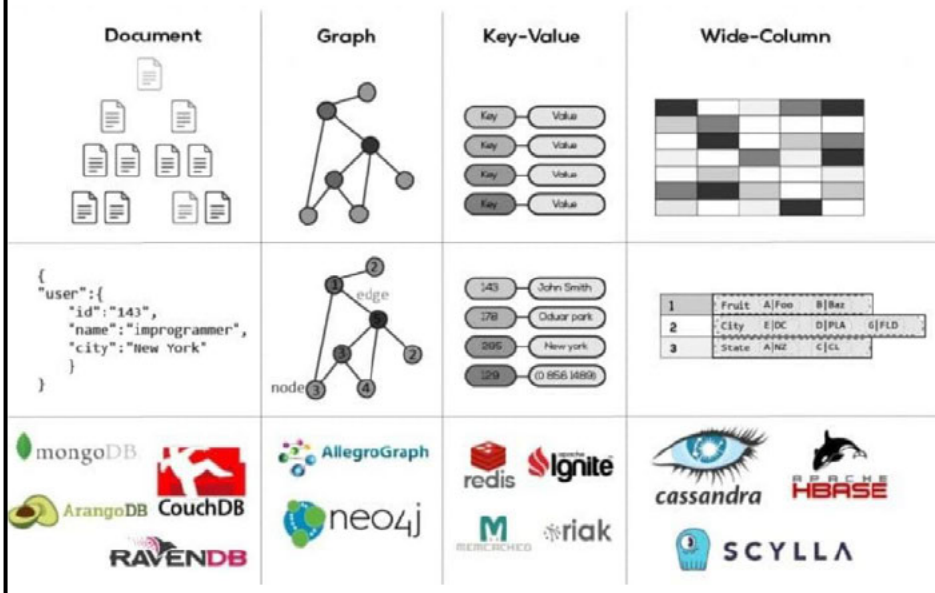
Rankings of DB Systems by Popularity

- NoSQL is moving up.
  - DB-Engines Ranking *by popularity* (https://db-engines.com/en/ranking May, 2021): From Websites, Google trends, social networks, job Ads, etc.

Rank May 2021	Rank Apr 2021	Rank May 2020	DBMS	Database Model
1.	1.	1.	Oracle	Relational, Multi-model
2.	2.	2.	MySQL	Relational, Multi-model
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model
4.	4.	4.	PostgreSQL	Relational, Multi-model
5.	5.	5.	MongoDB	Document, Multi-model
6.	6.	6.	IBM Db2	Relational, Multi-model
7.	7.	↑ 8.	Redis	Key-value, Multi-model
8.	8.	↓ 7.	Elasticsearch	Search engine, Multi-model
9.	9.	9.	SQLite	Relational
10.	10.	10.	Microsoft Access	Relational
11.	11.	11.	Cassandra	Wide column
12.	12.	12.	MariaDB	Relational, Multi-model
13.	13.	13.	Splunk	Search engine
14.	14.	14.	Hive	Relational
15.	15.	↑ 23.	Microsoft Azure SQL Database	Relational, Multi-model
16.	16.	16.	Amazon DynamoDB	Multi-model
17.	17.	↓ 15.	Teradata	Relational, Multi-model
18.	18.	↑ 20.	SAP HANA	Relational, Multi-model
19.	↑ 20.	↑ 21.	Neo4j	Graph

8

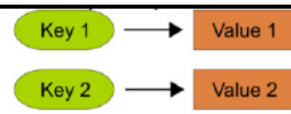
FOUR TYPES OF NOSQL DATABASES



://www.improgrammer.net/most-popular-nosql-database/

9

Key-Value Store



- A store of two fields: (Key, Value) pairs.
  - Value is a blob, needing parsing
- Only one way to access the data through hashing(Key) = value
- No multi-key transactions
- No query language! only get/put/delete/update/exec

Table T:

key	value
k1	v1
k2	v2
k3	v3
k4	v4

keys are sorted

10

Key-Value Store

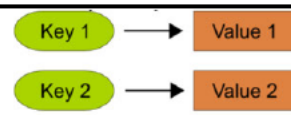


Table T:

key	value
k1	v1
k2	v2
k3	v3
k4	v4

keys are sorted

- API:
  - get(key) → value
  - get(key range) → values
  - getNext → value
  - put(key, value)
  - delete(key)
  - execute(key): Invoke an operation to a value.

11

NoSQL Models: Key/Value Stores

Employee

Key	Value		
01	Amanda	21	3.2
02	Benjamin	2.7	4.1
03	Chris	21	3.9
04	James	24	2.9
05	Amanda	19	2.9
06	Andy	23	3.5
07	Gabriel	24	8.0

Customer Profiles

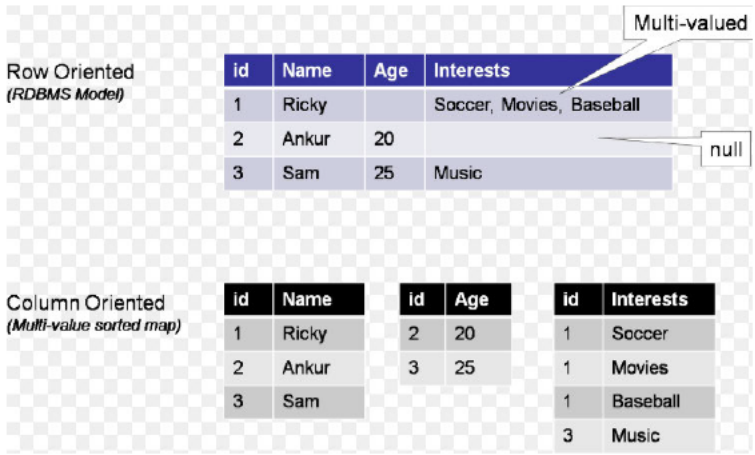
Key	Value
15740	'Morning Commuters', 'Come often, spend little. Stop in for a quick coffee and pastry in the morning', '25 - 45','Email'
23506	'Seasonal Celebrators', 'Middle-aged. A smaller segment, but our biggest source of revenue', '35 - 55', 'Email, Direct Mail'
23634	'One-time Weddings', 'Newlyweds, make up 15% of our revenue', '26 - 38', 'Email, Facebook, Instagram, Pinterest'

Blog

Key	Value
25	Big Data techniques offer several advantages over traditional polls. The large volume, wide variety, and high velocity of Big Data are the perfect ingredients to capture the political pulse. Volume increases the accuracy of the results and allow for better representation of opinions. Variety incorporates data sources from multiple political media sources in the form of text, images, or sounds. Velocity reflect the fast changing political climate, a necessary feature for political campaigns that want the latest results for better decision making. Big Data technologies are not only faster, but also relatively cheaper than traditional polls. Once the Big Data algorithms are constructed, they can run on relatively cheap platforms.
35	
36	The biggest advantage of using Big Data technologies for political campaigns is their ability to provide different views of the political spectrum. For example, sentiment analysis is a Big Data approach also known as opinion mining. The sentiment analysis can process natural language, analyze text, and perform computational linguistics to identify and extract subjective information in source materials.

12

Column Stores

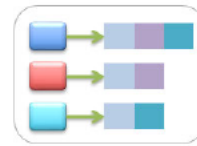


Row-Centric vs Column-Centric DB

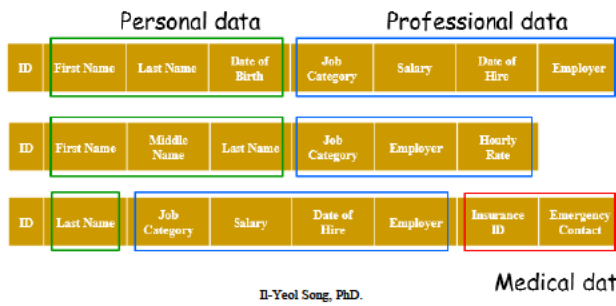
110,3195,Elaine,Castro,43511,62.3
103,4303,Monica,Hudson,43523,99.31
108,5440,Charlene,Rodriguez,43564,84.4
110,3311,Cody,McCormick,43626,71.54
109,3160,Matthew,Myers,43841,39.39
104,4653,Mack,Jennings,43873,80.44
104,2871,Frances,Huff,43918,55.16
...
103,56

110,103,108,110,109,104,104,...,103
3195,4303,5440,3311,3160,4653,2871,...,5699
Elaine,Monica,Charlene,Cody,Matthew,Mack,Frances,...,Yvette
Castro, Hudson, Rodriguez, McCormick, Myers, Jennings, Huff,..., Butler
43511,43523,43564,43626,43841,43873,43918,...,44189
62.3,99.31,84.4,71.54,39.39,80.44,55.16,...,26.44

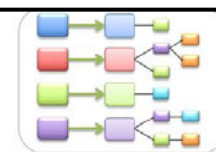
NoSQL Models: Column Store



- Motivated by Google's BigTable
- Extension of the K/V system, where columns can have a complex structure, rather than a blob value
  - Supports complex modeling structure (nested tables, repeating groups, set, list, etc.)



NoSQL Models: Document Store



- **Document Store**
  - Similar to Key-value store, but the value is a complete document, such as JSON, XML, etc.
  - Any collection of documents such as maps, collections, and scalar values.

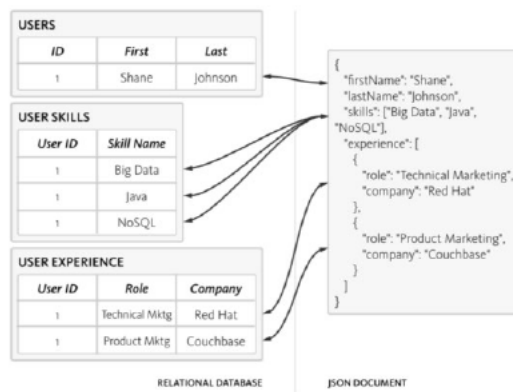
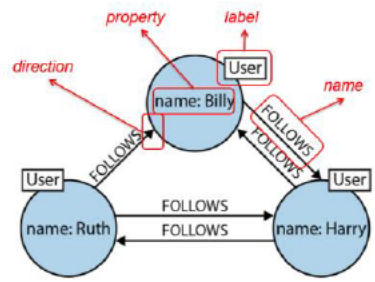


Figure 2: Multiple tables vs. nested data with JSON documents

NoSQL Models: Graph Databases

- **Graph Database**
  - Models data in terms of nodes and connections
  - Useful for inter-connected data such as communication patterns, social networks, bio interactions.
  - Allows us to ask deeper and more complex questions
  - Difficult to distribute components of a graph among a network of servers as graphs become larger.



Prospects on NoSQL Databases

- Most of them lack full ACID compliance for guaranteeing transactional integrity and data consistency.
  - ***Eventual consistency*** limits mission-critical transactional applications.
- RDBs and NoSQL DBs will co-exist for many years to come.
  - RDBs: Transaction-based systems
  - NoSQL: Search engines, web-based systems, real-time, cloud, mobile applications, low-cost initial engagement, IoT
  - Will add SQL-like language interface
- Weakness: Low-level language, no standards, administration, support, analytics, BI, No standards

MongoDB Shell

The following Mongo Shell provides a cloud based MongoDB engine. You can create database, insert/update records, queries, etc.

https://docs.mongodb.com/manual/tutorial/query-embedded-documents/

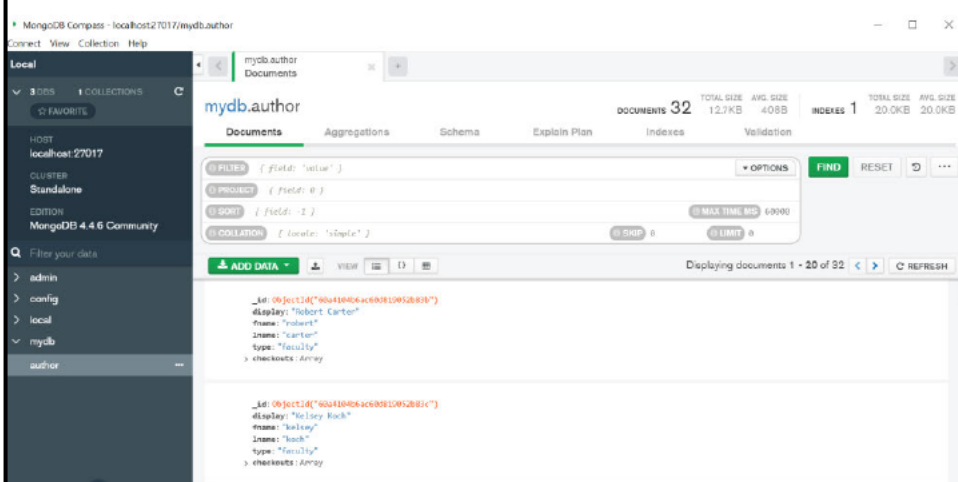
- The shell resets all data stored once closed.
- Create commands in Notepad and then copy and paste into the shell
- Be careful about quotes
- Up arrow display the previous command

MongoDB Installation

Version: 4.4.6 (May 2021):

- Install the ***Community version***
- See https://docs.mongodb.com/manual/installation/

MongoDB Compass



MongoDB Compass

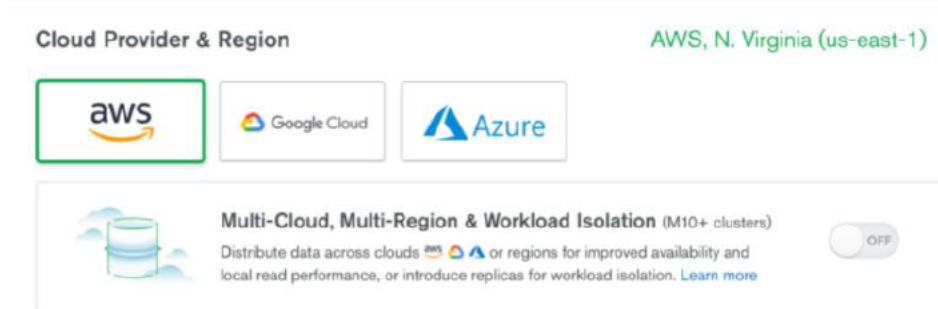
- A GUI tool for MongoDB
- Tutorials:

MongoDB Compass: Complete In-depth Tutorial [all features]
Jun 19, 2020
https://www.youtube.com/watch?v=ydXCcLAi5aU

MongoDB Compass Tutorial - Aggregation
Apr 9, 2020
Coco Zhu
https://www.youtube.com/watch?v=IqBXKRrgy38&t=10s

MongoDB Atlas

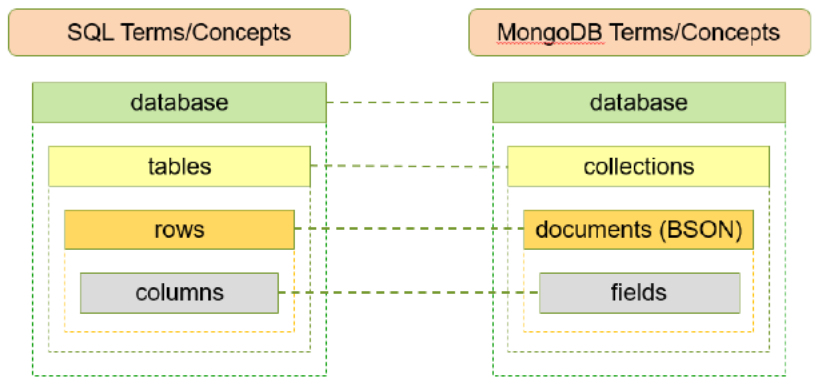
- A cloud-based MomgoDB cluster
- **https://www.mongodb.com/cloud/atlas**
- 512MB free storage



MongoDB Introduction

- A document-based NoSQL database by MongoDB, Inc.
- Released in 2009.
- Within MongoDB
  - **Data** is stored in **documents**.
  - **Documents** of a similar type are stored in **collections**.
  - Related **collections** are stored in a **database**.
- Documents are stored as **JSON** files, this makes it easier to read and manipulate using different programming languages.

RDBMS and MongoDB



JSON

- JavaScript Object Notation (JSON) – data interchange format used to represent data as a logical object.
- Every JSON document requires an object ID.
- **Object** is enclosed by a pair of **curly brackets as in {K:V}**
- **Array** is enclosed by a pair of **square brackets [ ]**

```
{_id: 101, title: "Database Systems", author: ["Coronel", "Morris"]}
```

- Example:

```
{
_id: 101,
title: "Database Systems",
author: ["Coronel", "Morris"]
}
```


NoSQL databases sacrifice redundancy to improve scalability

```
{
_id: 101,
title: "Database Systems",
author: ["Coronel", "Morris"],
publisher: {
        name: "Cengage",
        street: "500 Topbooks Avenue",
        city: "Boston",
        state: "MA"
        }
}
```

```
{
_id: 101,
title: "Database Systems",
author: [
        {
        name: "Coronel",
        email: "ccoronel@mtsu.edu",
        phone: "6155551212"
        },
        {
        name: "Morris",
        email: "smorris@mtsu.edu",
        office: "301 Codd Hall"
        }
        ],
publisher: {
    name: "Cengage",
    address: {
        street: "500 Topbooks Avenue",
        city: "Boston",
        state: "MA"
            }
        }
}
```

Syntactic Rules in MongoDB

- MongoDB is ***Case-Sensitive*** – Capitalization matters.
- MongoDB uses **camelCase** as in iPhone, eBay, dailyJumpRate
- Semi-colons are not required.
- **All string data** should be in ***double or single quotes***.
- Commands are **space-independent**.
  - Example:-

The compiler ignores spaces

```
> db.Employee.update(
... { "Employeeid" : 2},
... { $set: {"Employeeid" : "NewMartin"}});
```

- Comments are indicated by //

```
//Retrieve comments by user
db.comments.find({'_userId' : someUserId})
```

- Data is displayed in the order of the insertion order.
- The field names **cannot** start with the **$** character.
- The field names **cannot** contain the **.** character.
- Use proper indentation and pretty() method when printing

MongoDB
CRUD Operations

Create a Database

- **Database** – It is a container for collections. A MongoDB server can store multiple databases. Each database has its own group of files.
- **Create a database command** – **"use"**
  - Creates a new database if a database in that name doesn't exist. Once created, it switches to the created database.

use <database_name>

```
> use EmployeeDB
```

Name of the Database

Output - Database is created

```
switched to db EmployeeDB
```


Create a collection

- To create a collection, use the method – **createCollection()**
- Use the **db** variable with the above method,
- Example –

db.createCollection("newproducts")

  - Creates a collection named "newproducts" inside the previously defined demo database.

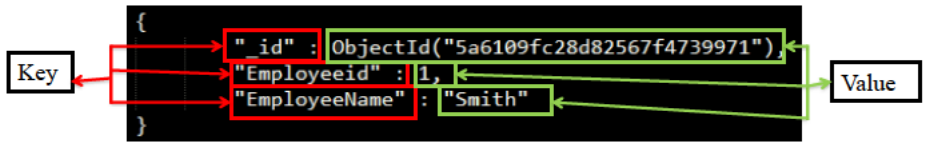
```
> use demo
switched to db demo
> db.createCollection("newproducts")
{ "ok" : 1 }
```

  - As the database contains at least one collection, it is shows using the **"show dbs"** command.

```
> show collections
newproducts
```

Common Terms in MongoDB

- **Field** – A key-value pair in a document is denoted as a field. **A key name may be written with or without quotes. While text data entered as value should be written in quotes.** (key: value, key: value, ..}



- **_id**
  - A mandatory field for every document.
  - Serves as the ***primary key*** of the document.
  - If you do not assign a value to this variable, MongoDB will automatically assign a value.

CRUD Operations: Create/Insert

- Create/Insert
  - db.collection.insert(<document>)
  - db.collection.insertOne(<document>)
  - db.collection.insertMany([{d1}, {d2}.., {d3}])
  - db.collection.save(<document>) //deprecated

```
> db.user.insert({
      first: "John",
      last : "Doe",
      age: 39
})
```

CRUD: Inserting documents

- Example:- **db.<<collectionName>>.insertOne({document})**

```
db.users.insertOne(          ← collection
   {
      name: "sue",           ← field: value
      age: 26,               ← field: value   } document
      status: "pending"      ← field: value
   }
)
```

```
>>> db.emp.insert({id: 1, name: "Song"})
WriteResult({ "nInserted" : 1 })
>>>
```


CRUD: Inserting Multiple documents

- Use the method – **insertMany([{d1}, {d2}.., {dn}])**
- Example:-

```
db.inventory.insertMany([
  { item: "journal", qty: 25, tags: ["blank", "red"], size: { h: 14, w: 21, uom: "cm" } },
  { item: "mat", qty: 85, tags: ["gray"], size: { h: 27.9, w: 35.5, uom: "cm" } },
  { item: "mousepad", qty: 25, tags: ["gel", "blue"], size: { h: 19, w: 22.85, uom: "cm" }
}
])
```

CRUD Operations: Read

- Read
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)

```
> db.user.find ()
{
        "_id" : ObjectId("51…"),
        "first" : "John",
        "last" : "Doe",
        "age" : 39
}
```


READ Operation: pretty() method

- To improve the readability of the retrieved document, use the method – **pretty()**
- Example:-

```
> db.products.find()
{ "_id" : ObjectId("598e01613ae3ad8abf1b8300"), "name" : "standard desk chair", "price" : 150, "brand" : "CheapCo", "type" : "chair" }
> db.products.find().pretty()
{
        "_id" : ObjectId("598e01613ae3ad8abf1b8300"),
        "name" : "standard desk chair",
        "price" : 150,
        "brand" : "CheapCo",
        "type" : "chair"
}
```

READ with find() method

- Syntax - **find({<query>}, {<projection>})**
  - Both objects are optional
  - ***If only one object parameter is written, MongoDB assumes it belongs to the query object parameter***
  - **{<query>} is the same as WHERE clause**
  - **{<projection>} is the same as SELECT clause**
  - **find({<WHERE>}, {<SELECT>})**

```
SELECT name, cell
FROM  friends
WHERE City = "Boston"
```

```
db.friends.find(
        {City : "Boston"},
        {name:1, cell: 1, _id:0 }
)
```


READ with find() method

- Syntax - **find({<query>}, {<projection>})**

```
db.users.find(                          <-- collection
   { age: { $gt: 18 } },                <-- query criteria
   { name: 1, address: 1 }              <-- projection
).limit(5)                              <-- cursor modifier
```

- **The value with each key in the projection object** is either the value 0 or 1.
  - **'1'** –the key:value pair should be included in the results.
  - **'0'** –the key:value pair should be omitted in the results.
- **Retrieve the *name* and *cell#* fields and suppress _id field**

  SELECT name, cell FROM patron;

  db.patron.find({}, {name:1, cell: 1, _id:0})

find() method w/o Query Object

- Syntax - **find({<query>}, {<projection>})**
  - When projection object is needed without a query object, an empty object must be used as a query object

- **Retrieve the display field of every document**

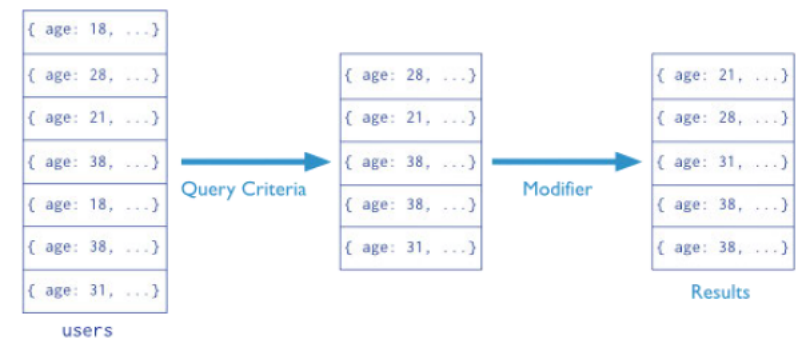
  SELECT name, cell FROM friends;

  db.patron.find({}, {name:1, cell: 1, _id:0})

find() with Query Criteria

Collection — Query Criteria — Modifier

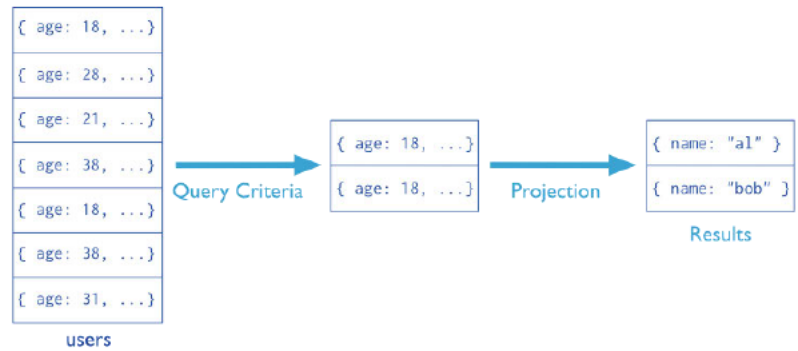
```
db.users.find( { age: { $gt: 18 } } ).sort( {age: 1 } )
```



Find() with Query and Projections

Collection Query Criteria Projection

```
db.users.find( { age: 18 }, { name: 1, _id: 0 } )
```



Comparison Operators

Name	Description
$eq	Matches values that are equal to a specified value.
$gt	Matches values that are greater than a specified value.
$gte	Matches values that are greater than or equal to a specified value.
$lt	Matches values that are less than a specified value.
$lte	Matches values that are less than or equal to a specified value.
$ne	Matches all values that are not equal to a specified value.
$in	Matches any of the values specified in an array.
$nin	Matches none of the values specified in an array.

https://docs.mongodb.com/manual/reference/operator/query-comparison/

Logical Operators

Name	Description
$or	Joins query clauses with a logical OR returns all documents that match the conditions of either clause.
$and	Joins query clauses with a logical AND returns all documents that match the conditions of both clauses.
$not	Inverts the effect of a query expression and returns documents that do *not* match the query expression.
$nor	Joins query clauses with a logical NOR returns all documents that fail to match both clauses.

https://docs.mongodb.com/manual/reference/operator/query-logical/

SQL and MongoDB: SELECT

```
SELECT * FROM users WHERE age>33
db.users.find({age:{$gt:33}})

SELECT * FROM users WHERE age!=33
db.users.find({age:{$ne:33}})

SELECT * FROM users WHERE a=1 and b='q'
db.users.find({a:1,b:'q'})

SELECT * FROM users WHERE a=1 or b=2
db.users.find( { $or : [ { a : 1 } , { b : 2 } ] } )

SELECT * FROM foo WHERE  name='bob' and (a=1 or b=2 )
db.foo.find( { name : "bob" , $or : [ { a : 1 } , { b : 2 } ] } )

SELECT * FROM users WHERE age>33 AND age<=40
db.users.find({'age':{$gt:33,$lte:40}})
```

- **MongoDB operators start with $.**
- Query operators:
  - $ne, $gt, $gte $lt, $lte,
  - $and, $or, $not
  - $exists, and $regex

SQL and MongoDB: aggregation: count()

```
SELECT COUNT(*)
FROM users
```

```
db.users.count()
```

```
db.users.find().count()
```

```
SELECT COUNT(user_id)
FROM users
```

```
db.users.count( { user_id: { $exists: true } } )
```

```
db.users.find( { user_id: { $exists: true } } ).count()
```

```
SELECT COUNT(*)
FROM users
WHERE age > 30
```

```
db.users.count( { age: { $gt: 30 } } )
```

```
db.users.find( { age: { $gt: 30 } } ).count()
```

47

47

SQL and MongoDB: aggregation: distinct()

```
SELECT DISTINCT(status)
FROM users
```

```
db.users.distinct( "status" )
```

48

48

SQL and MongoDB: ORDER BY

```
SELECT *
FROM users
WHERE status = "A"
ORDER BY user_id ASC
```

In sort() method:
(-1) stands for descending order
(1)stands for ascending order

```
db.users.find( { status: "A" } ).sort( { user_id: 1 } )
```

```
SELECT *
FROM users
WHERE status = "A"
ORDER BY user_id DESC
```

```
db.users.find( { status: "A" } ).sort( { user_id: -1 } )
```


Sorting the output: find() with sort()

Example – *retrieves the display name and age from documents for patrons that are age 30 or less and sorts them in descending order of age.*

SELECT display, age FROM patron WHERE age <= 30
ORDER BY age DESC;

- **db.patron.find({age: {$lte:30} }, {display:1, age:1}).sort({age: −1})**

```
> db.patron.find({age: {$lte:30} }, {display:1, age:1}).sort({age: -1});
{ "_id" : ObjectId("598e0649b4615ba6815141ce"), "display" : "Jimmie Love", "age" : 29 }
{ "_id" : ObjectId("598e0649b4615ba6815141dd"), "display" : "Desiree Harrington", "age" : 28 }
{ "_id" : ObjectId("598e0649b4615ba6815141d6"), "display" : "Keith Cooley", "age" : 27 }
{ "_id" : ObjectId("598e0649b4615ba6815141d1"), "display" : "Holly Anthony", "age" : 25 }
{ "_id" : ObjectId("598e0649b4615ba6815141d4"), "display" : "Iva Ramos", "age" : 24 }
{ "_id" : ObjectId("598e0649b4615ba6815141ea"), "display" : "Betsy Malone", "age" : 24 }
{ "_id" : ObjectId("598e0649b4615ba6815141d5"), "display" : "Rena Mathis", "age" : 23 }
{ "_id" : ObjectId("598e0649b4615ba6815141e2"), "display" : "Zach Kelly", "age" : 23 }
{ "_id" : ObjectId("598e0649b4615ba6815141e5"), "display" : "Wilfred Fuller", "age" : 23 }
{ "_id" : ObjectId("598e0649b4615ba6815141e6"), "display" : "Jeff Owens", "age" : 23 }
{ "_id" : ObjectId("598e0649b4615ba6815141e7"), "display" : "Homer Goodman", "age" : 23 }
{ "_id" : ObjectId("598e0649b4615ba6815141eb"), "display" : "Tony Miles", "age" : 23 }
```

age:-1 //descending order
age: 1 //ascending order

Method Chaining: sort(), limit(), skip()

- Sort(), limit(), skip() can be chained in any order
- The limit() method restricts the number of documents returned.

```
SELECT *
FROM users
LIMIT 1
```

```
db.users.findOne()
```

```
db.users.find().limit(1)
```

findOne() that returns only the first document.
db.patron.findOne({type: "student"})
is the same as
db.patron.find({type: "student"}).limit(1)

51

Method Chaining: skip()

The skip() method is used for skipping the given number of documents in the Query result.

```
SELECT *
FROM users
LIMIT 5
SKIP 10
```

```
db.users.find().limit(5).skip(10)
```

The limit(5).skip(10) methods skips 10 documents and then displays the next 5 documents.

52

CRUD Operations: Update

- Update operations modify existing documents in a collection (Default: a single document)
  - db.collection.update(<query>, <update>, <options>)
  - db.collection.updateOne(<query>, <update>, <options>)
  - db.collection.updateMany(<query>, <update>, <options>)
  - db.collection.replaceOne(<query>, <replacement>, <options>)

- **updateOne()** updates some fields.
- **replaceOne()** replace the entire document

```
db.users.updateMany(                          <-- collection
   { age: { $lt: 18 } },                      <-- update filter
   { $set: { status: "reject" } }             <-- update action
)
```


UPDATE Operation

- **Multifields update**

```
> db.users.update(
        { age: {$gt: 60}},
        { $set: {
                bonus: 1000,
                salary: 7000}
        }
)
```

SQL and MongoDB: UPDATE

```
UPDATE people
SET status = "C"
WHERE age > 25
```

```
db.people.updateMany(
   { age: { $gt: 25 } },
   { $set: { status: "C" } }
)
```

```
UPDATE users
SET age = age + 3
WHERE status = "A"
```

```
db.users.update(
   { status: "A" } ,
   { $inc: { age: 3 } },
   { multi: true }
)
```

```
db.users.updateMany(
   { status: "A" } ,
   { $inc: { age: 3 } }
)
```


CRUD Operations: Delete

- Delete: remove documents from a single collection:
  - db.collection.deleteOne(<query>, <options>)
  - db.collection.deleteMany(<query>, <options>)
  - db.collection.deleteMany({}) //delete all documents
  - db.collection.remove(<query>, <justOne>) //old method

```
db.users.deleteMany(          <-- collection
   { status: "reject" }       <-- delete filter
)
```

SQL and MongoDB: DELETE

```
DELETE FROM users
WHERE status = "D"
```

```
db.users.remove( { status: "D" } )
```

- Remove the first document from the **users** collection where the **status** field equals "D":

  db.users.deleteOne({ status: "D" })

- Remove all documents from the **users** collection where the **status** field equals "A":
  db.users.deleteMany({ status : "A" })

DELETE ALL Operation

- To delete all the documents from a collection, pass an empty filter document {} to deleteMany():
- The following example deletes all documents from the users collection:
  db.users.deleteMany({})
- Using remove({})

```
> db.users.remove({})
WriteResult({ "nRemoved" : 2 })
```

Aggregations in MongoDB by Example

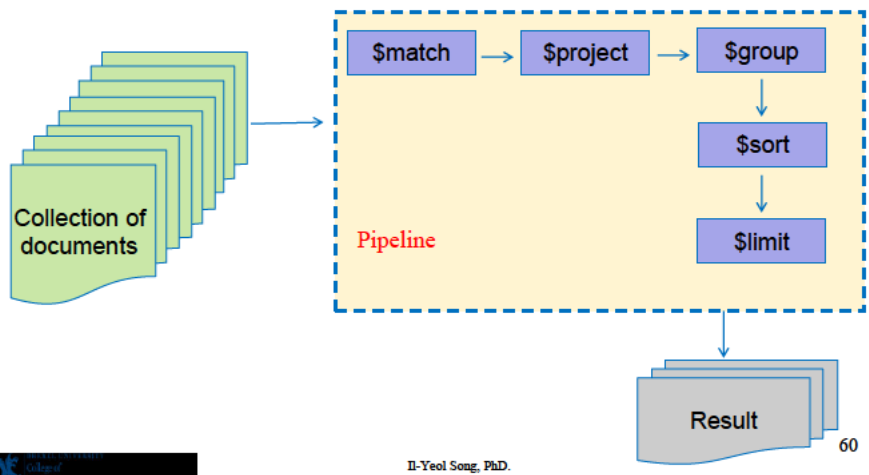
https://www.compose.com/articles/aggregations-in-mongodb-by-example/

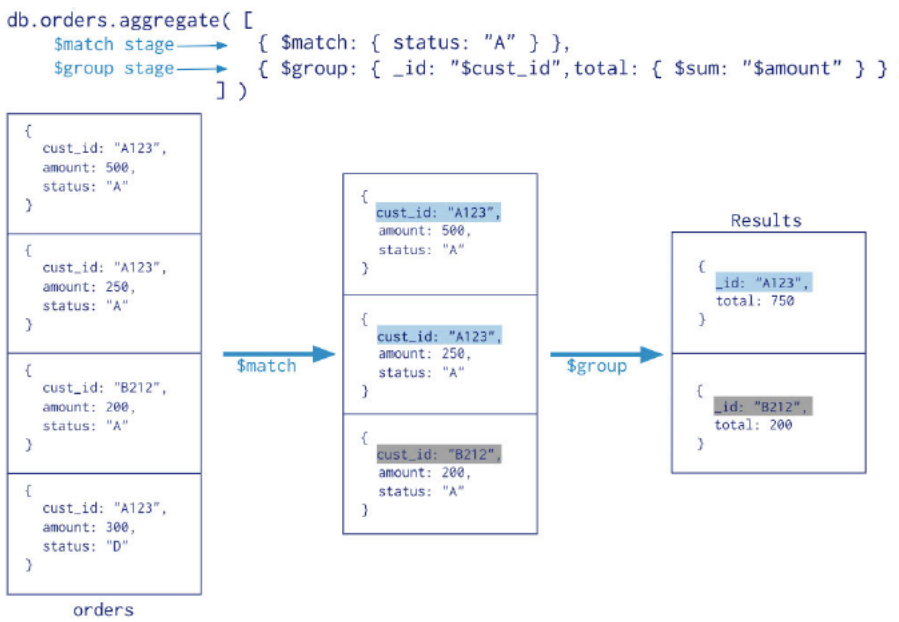
Aggregation Pipeline

https://docs.mongodb.com/manual/core/aggregation-pipeline/

Aggregation Pipeline Framework in MongoDB

- A framework for data aggregation through a data processing pipelines.
- Documents enter a multi-stage pipeline that transforms the documents into aggregated results.





61

Comparison b/n SQL and Aggregation in MongoDB

SQL Terms	MongoDB Aggregation Operators
WHERE	$match
GROUP BY	$group
HAVING	$match
SELECT	$project
ORDER BY	$sort
LIMIT	$limit
SUM()	$sum
COUNT()	$sum

62

Example Collection

A ***collection*** includes ***documents*** called ***transactions***

```
{
  "id":
  "produ
  "custc
  "amoun
  "trans
}
```

```
{
  "id": "
  "produc
  "custom
  "amount
  "transa
}
```

```
{
  "id": "1",
  "productId": "1",
  "customerId": "1",
  "amount": 20.00,
  "transactionDate": ISODate("2017-02-23T15:25:56.314Z")
}
```


63

Gaining Insights with Sum, Min, Max, and Avg

- The monthly metrics with the average price of each transaction, and the minimum and maximum transaction in the month:

```
> db.transactions.aggregate([
  {
    $match: {
      transactionDate: {
        $gte: ISODate("2017-01-01T00:00:00.000Z"),
        $lt: ISODate("2017-01-31T23:59:59.000Z")
      }
    }
  }, {
    $group: {
      _id: null,
      total: { $sum: "$amount" },
      average_transaction_amount: { $avg: "$amount"},
      min_transaction_amount: { $min: "$amount" },
      max_transaction_amount: { $max: "$amount"}
    }
  }
]);
```

Output

```
{
  _id: null,
  total: 20333.00,
  average_transaction_amount: 8.50,
  min_transaction_amount: 2.99,
  max_transaction_amount: 347.22
}
```


64

Other Topics

- Pattern matching
- Array manipulation
- Indexing in MongoDB
- Sharding in MongoDB
- PyMongo

65

Question?

66