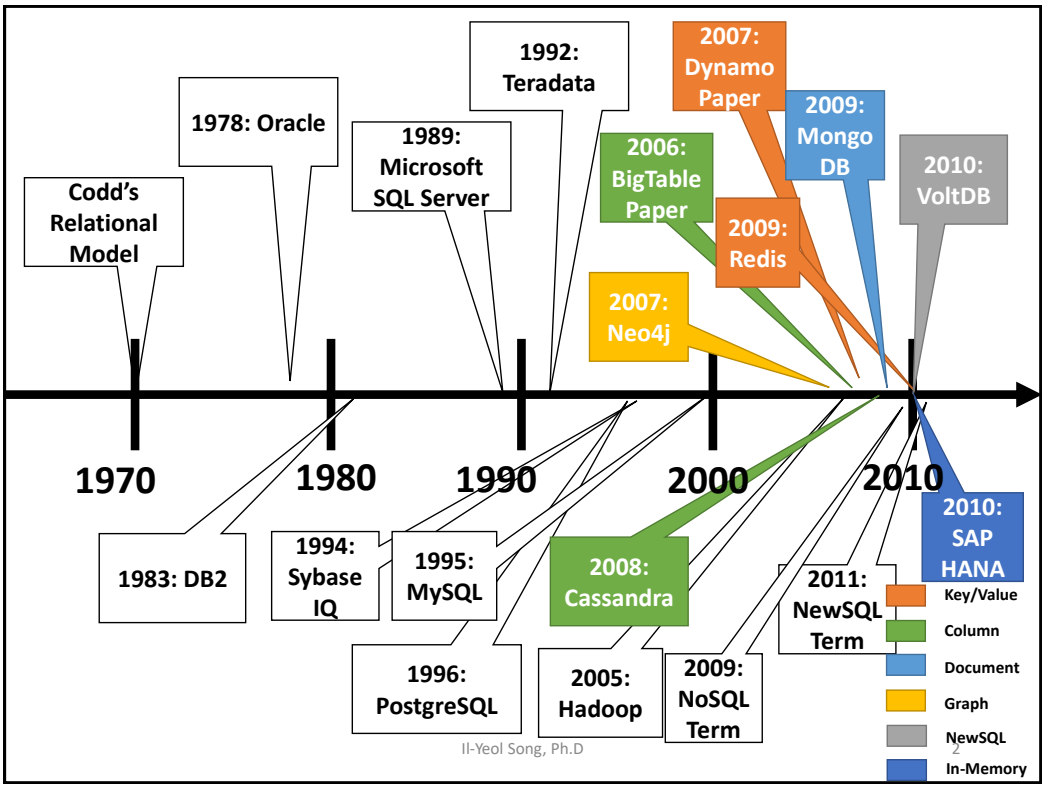
LEADS 3-Day Camp
Session 2: Big Data Management
(2) NoSQL Databases and MongoDB

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/



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

Challenges to RDBs

- New applications: e-commerce, social networks, IOTs, cloud
- Explosion of data (Volume)
- Globally distributed systems
  - Replication
  - Scalability
- Specialized applications: IOT, sensors, real-time applications: (Velocity)
- Different types of data (Variety)

=> Non-relational, scalable, flexible systems

=> Needs of systems with **big data**, **big users**, and **cloud** in mind

What's Wrong with RDBMSs?

- Some large applications require very high throughput (emails, shopping carts, stocks, Ads, IoT applications, cloud applications, etc)
- Rigid Schema:
  - RDBMSs need schema before writing—too rigid and slow
  - Rigid schema waste spaces for very sparse data
- ACID properties are too strict for some applications
  - Some apps need availability even with a network failure
  - Many background processes for transaction reliability
- Not all data is relational (impedance mismatch):
  - Complex data, JSON, array data, graph data
- Hence:
  - RDBMSs have problems with 3Vs: too big, too fast, and too diverse

NoSQL Databases

- NoSQL = Non-Relational
  - NoSQL: Originally "NO SQL", later "Not only SQL"
  - Support unstructured or **non-relational data types** (graph, document, JSON, etc.)
  - **Schema-less**: no rigid schema enforced by the DBMS
    - RDB: Schema-on-write
    - **NoSQL: Schema-on-read**
  - **Scale out massively** at low cost and fast retrieval (elastic scaling)
  - **Low cost operational** data management for large #users
  - Support **scalability, performance, fault-tolerance**
  - May not guarantee full **ACID properties**
  - Designed for **real-time, non-uniform, big data**.

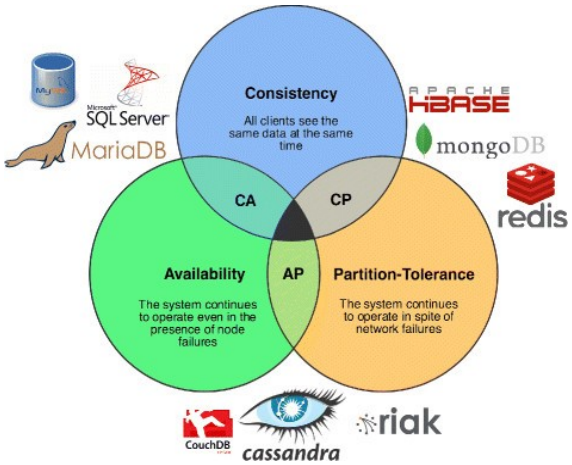
CAP Theorem

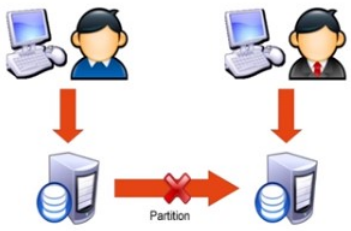
In a distributed system, we can choose only two out of the following three guarantees (Eric Bower, 2002)

- **Consistency**: Every read receives the most recent value
- **Availability**: Every request receives a (non-error) response – without the guarantee that it contains the most recent write. no downtime.
- **Partition Tolerance**: The system continues to operate despite an arbitrary number of nodes being dropped.



CAP Theorem and NoSQL Databases





Rankings of DB Systems by Popularity

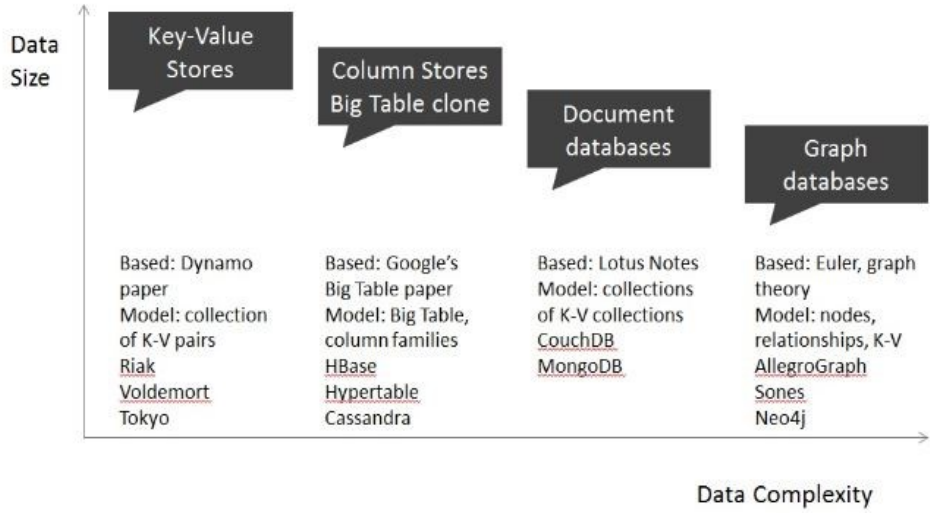
- NoSQL is moving up.
  - DB-Engines Ranking *by popularity* (https://db-engines.com/en/ranking May, 2019)

Rank May 2019	Rank Apr 2019	Rank May 2018	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
6.	6.	6.	IBM Db2	Relational, Multi-model
7.	↑ 8.	↑ 9.	Elasticsearch	Search engine, Multi-model
8.	↓ 7.	↓ 7.	Redis	Key-value, Multi-model
9.	9.	↓ 8.	Microsoft Access	Relational
10.	↑ 11.	10.	Cassandra	Wide column
11.	↓ 10.	11.	SQLite	Relational
12.	12.	↑ 14.	MariaDB	Relational, Multi-model
13.	13.	13.	Splunk	Search engine
14.	↑ 15.	↑ 18.	Hive	Relational
15.	↓ 14.	↓ 12.	Teradata	Relational

College of Computing & Informatics

9

NOSQL DATABASES

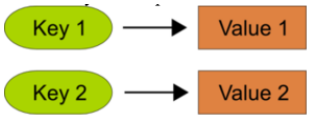


Source: http://www.smart421.com/big-data-2/mystery-solved-no-sql-means-not-only-sql/

College of Computing & Informatics

Il-Yeol Song, Ph.D | 10

NoSQL Models: Key/Value Systems



- A store of (Key, Value) pairs.
- Only one way to access the data through h(Key) = value
- No query language: only get/put/delete/update

Key	Value
6.01	"Intro to EECS 1 by Professor Madden"
6.02	"Intro to EECS 2 by Professor Stonebraker"
6.033	"Systems by Professor Zeldovich"
6.814	"Databases by Professor Smith"

Key Value Representation of Course Catalog

Programming Language is get/put

get("6.01") ➔ "Intro to EECS 1 by Professor Madden"
put("6.005","Software Engineering by Prof. Jones")

Limited multi-record transactional consistency

NoSQL Models: Key/Value Systems

- K/V models are expanded to complex columns and documents such as XML/JSON

Key	Value
6.01	{title: "Intro To EECS", prof: "Madden", room: 123}
6.02	{title: "Intro To EECS 2", professor: "Stonebraker"}
6.033	{title: "Systems", room: 145}
6.814	{title: "Databases, room: 154, professor: "Smith"}

Different fields in docs

Document Representation of Course Catalog

Can lookup documents by key
Also some ability to search contents of documents

Typically, no joins or multi-document updates

Column Stores

Row Oriented
(RDBMS Model)

id	Name	Age	Interests
1	Ricky		Soccer, Movies, Baseball
2	Ankur	20	
3	Sam	25	Music

Multi-valued

null

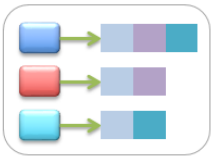
Column Oriented
(Multi-value sorted map)

id	Name
1	Ricky
2	Ankur
3	Sam

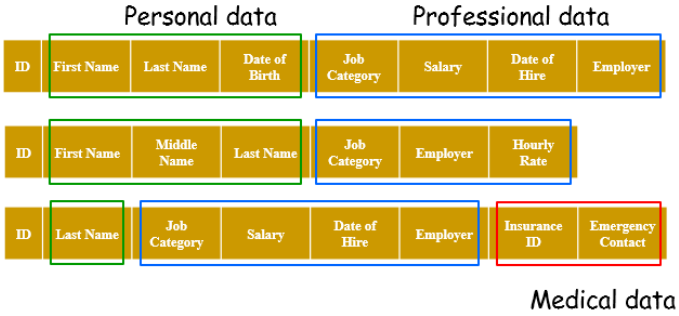
id	Age
2	20
3	25

id	Interests
1	Soccer
1	Movies
1	Baseball
3	Music

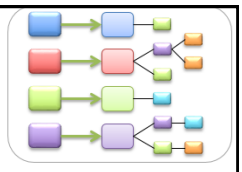
NoSQL Models: Column Store



- 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.

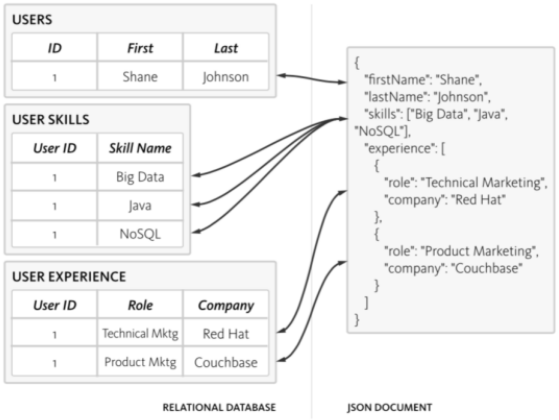
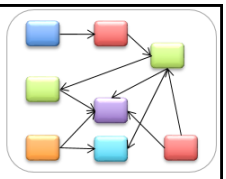
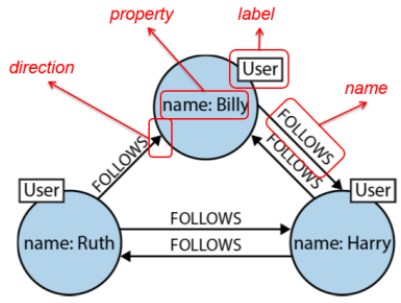


Figure 2: Multiple tables vs. nested data with JSON documents

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.



NewSQL

- NewSQL is a class of *new relational database* with *scale-out scalability* and *performance*:
  - provide the **scalable performance** comparable to NoSQL systems for OLTP workloads
  - Support SQL and the **ACID** properties
  - Ex: Google spanner, VoltDB, MemSQL, NuoDB, Clustrix

NewSQL

- Two common features of various NewSQL systems
  - They all support the relational data model (manage both **structured** and **unstructured**)
  - They all use SQL as their primary interface.
- Weakness: Limited support for "variety" due to the need of schema-on-write

	Old SQL	NoSQL	NewSQL
Relational	Yes	No	Yes
SQL	Yes	No	Yes
ACID transactions	Yes	No	Yes
Horizontal scalability	No	Yes	Yes
Performance / big volume	No	Yes	Yes
Schema-less	No	Yes	No

Source: http://labs.sogeti.com/newsql-whats/

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 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.

JSON Documents

- JavaScript Object Notation (JSON) – data interchange format used to represent data as a logical object.
- JSON Objects are enclosed in curly brackets {} that contain **key-value** pairs.
- Format used – **key:value**
- Example –

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

  - They are also displayed in the below format –

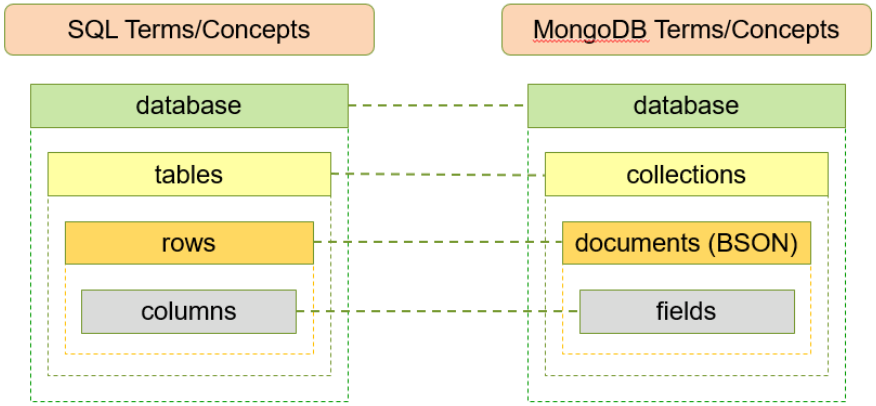
```
{
_id: 101,
title: "Database Systems",
author: ["Coronel", "Morris"]
}
```


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"
            }
        }
}
```

RDBMS and MongoDB



Syntactic Rules in MongoDB

- MongoDB is Case-Sensitive – Capitalization matters.
- Semi-colons are not required.
- All string data being saved should be in double quotes.
- Commands are space-independent.
  - Example:-

The compiler ignores spaces

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

- MongoDB follows natural order of the insertion order.
  - Any record entered first will be displayed first while using the print() function, unless specified explicitly using the sort command.

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.

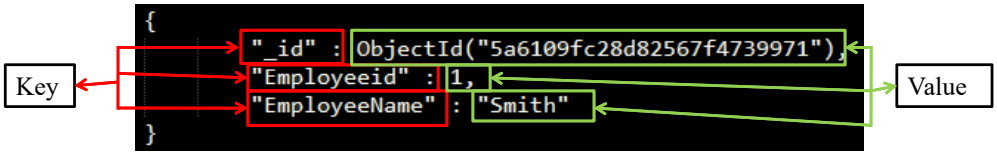
```
> use demo
switched to db demo
> db.createCollection("newproducts")
{ "ok" : 1 }
```

  - 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. **All key names should be written in quotes**. **While text data entered as value should be written in quotes**.



- **_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.

Inserting documents

- Use the method – **insert()**
- Documents details are the parameter for the method.
- Syntax for insertion –

db.<collection name>.insert({document})

- Example:-

```
db.products.insert ({name: "standard desk chair",
        price: 150,
        brand:  "CheapCo",
        type: "chair"})
```

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

SQL and MongoDB: CREATE/INSERT

```
CREATE TABLE users (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```

Implicitly created on first `insert` operation. The primary key `_id` is automatically added if `_id` field is not specified.

```
db.users.insert( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } )
```

However, you can also explicitly create a collection:

```
db.createCollection("users")
```

```
db.people.insertOne(
   { user_id: "bcd001", age: 45, status: "A" }
)
```


SQL and MongoDB: DROP/INDEX

```
DROP TABLE users
```

```
db.users.drop()
```

```
CREATE INDEX idx_user_id_asc
ON users(user_id)
```

```
db.users.ensureIndex( { user_id: 1 } )
```

```
CREATE INDEX
       idx_user_id_asc_age_desc
ON users(user_id, age DESC)
```

```
db.users.ensureIndex( { user_id: 1, age: -1 } )
```

Document Retrieval

- To retrieve and display the documents present in the collection, use the method – **find()**
- Example:-

```
> db.products.find()
```

  - Output:-

```
{ "_id" : ObjectId("598e01613ae3ad8abf1b8300"), "name" : "standard desk chair",
"price" : 150, "brand" : "CheapCo", "type" : "chair" }
```

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

```
> db.products.find().pretty()
```

  - Output:-

```
{
        "_id" : ObjectId("598e01613ae3ad8abf1b8300"),
        "name" : "standard desk chair",
        "price" : 150,
        "brand" : "CheapCo",
        "type" : "chair"
}
```

College of Computing & Informatics

Selection and Restriction 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**
  - When the query object is not needed but projection object is needed, an empty object must be used as a query object

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

  SELECT display FROM patron;

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

College of Computing & Informatics

Querying Documents in MongoDB

```
SELECT *
FROM
   [users]
WHERE
  name = "Johan A"
```

```
db.users.find
  (
   {
      name: "Johan A"
   }
  )
```


Selection and Restriction in find()

- Find display and type of "Robert Carter"

SELECT display, type FROM patron
WHERE display = "Robert Carter";

db.patron.find({display: "Robert Carter"}, {display:1, type:1})

- Find display of "Faculty" patron

SELECT display FROM patron WHERE type = "faculty";

db.patron.find({type: "faculty"}, {display:1, _id:0})

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 $.**
- Other Query operators:
  - $ne, $gt, $gte $lt, $lte,
  - $and, $or,
  - $exists, and $regex.

SQL and MongoDB: Aggregation

```
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()
```

SQL and MongoDB: Aggregation

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

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


SQL and MongoDB: ORDER BY

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

```
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 } )
```

SQL and MongoDB: Limit

```
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)

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

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


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 } }
)
```

SQL and MongoDB: DELETE

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

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

```
DELETE FROM users
```

```
db.users.remove( )
```


Pattern Matching

- Define the search criteria using the **$regex** operator.
- Or, use **"/pattern/"** in place of the $regex operator. This is known as a delimiter. We specify the pattern we are looking for in-between the delimiters.

Using regex Expression

The following regex query searches for all the posts containing string **tutorialspoint** in it:

```
>db.posts.find({post_text:{$regex:"tutorialspoint"}})
```

The same query can also be written as:

```
>db.posts.find({post_text:/tutorialspoint/})
```

Pattern Matching with wild characters

Wild Characters:

'^' character denotes that the string starts with a specific character, while '$' denoted that the string ends with a specific character.

- Find posts which has a tag that begins with only "tut"

  SELECT * FROM posts WHERE tags LIKE "tut%";

  db.posts.find({tags: {$regex: "**^tut**"}})

- Find posts which ends with "MongoDB" in the post_text

SELECT * FROM posts WHERE post_text LIKE "%MongoDB";

db.posts.find({post_text: {regex: "/MongoDB$/"}})

SQL and MongoDB: String Manipulation

```
SELECT *
FROM users
WHERE user_id like "%bc%"
```

```
db.users.find(
   { user_id: /bc/ }
)
```

db.users.find({ user_id: { $regex: /bc/ } })

```
SELECT *
FROM users
WHERE user_id like "bc%"
```

```
db.users.find(
   { user_id: /^bc/ }
)
```

db.users.find({ user_id: { $regex: /^bc/ } })

Aggregations in MongoDB by Example

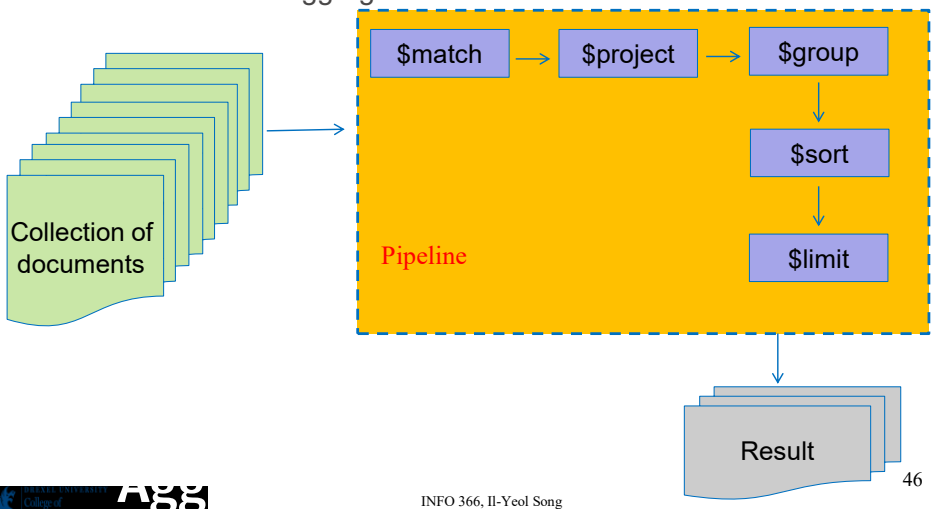
https://www.compose.com/articles/aggregations-in-mongodb-by-example/

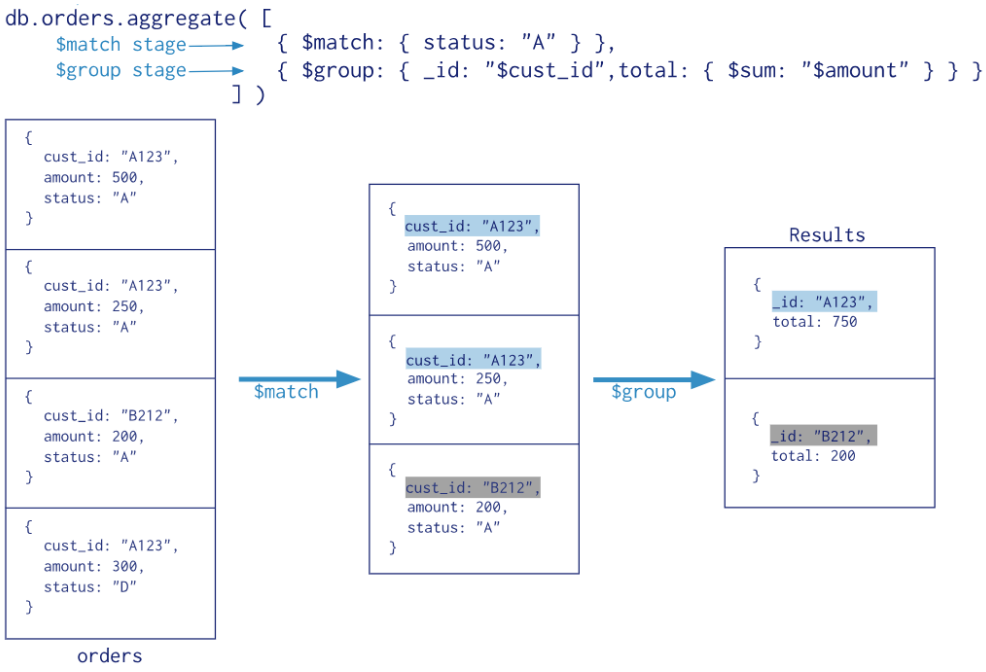
Aggregation Pipeline

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.





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

Example Collection

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

```
{
  "id":
  "produ
  "custo
  "amour
  "trans
}
```

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

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


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
}
```

MongoDB Integration

- MongoDB site: Webinars, free online courses, News, etc.
  http://www.10gen.com/
  http://www.mongodb.org/
  - Documentation, Downloads
- **Hadoop**
  https://github.com/mongodb/mongo-hadoop
- **Storm**
  https://github.com/christkv/mongo-storm
- **Spark**
  https://www.mongodb.com/products/spark-connector

MongoDB Shell

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

- The below shell resets all data stored once closed.

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

- Copy from Notepad and paste into the shell
- Be careful about quotes
- Up arrow display the previous command

- Various other Database commands can be found on the following website –

  https://docs.mongodb.com/manual/reference/command/

- Watch the below Youtube video for a better understanding on how to code with MongoDB –

https://www.youtube.com/watch?v=pWbMrx5rVBE&index=1&list=PLkkZhH5Zlk0xwDXgOaNN8TH8X0ySnwAG0&t=776s

Question?