NoSQL Databases Part 2 : Week 5

NoSQL

MongoDB – Further querying

NoSQL Vs Relational DBMS

Why NoSQL?

- **Massive Scalability:** Handles tens of thousands to millions of users without slowing down.
- **Always Available:** Designed to run continuously, so there's no downtime.
- **Flexible Data Handling:** Works well with both structured data (like tables) and unstructured data (like text or JSON).
- **Fast Response Times:** Provides quick responses, giving users a smooth experience.
- **Global Reach:** Easily supports users from around the world by replicating data in multiple regions.
- **Agile Development:** Allows frequent updates and changes to meet evolving business needs.
- **Cost-Effective Growth:** Easily adds more servers to scale up without huge extra costs.

Who uses NoSQL?

- Example Global 2000 enterprises that are using NoSQL for mission-critical systems:
  - **Tesco**, Europe's No.1 retailer, deploys NoSQL for e-commerce, product catalog, and other applications
    - https://www.couchbase.com/customers/tesco
  - **Ryanair**, the world's busiest airline, uses NoSQL to power its mobile app serving over 3 million users
    - https://www.computerworlduk.com/data/ryanair-invests-in-nosql-mobile-platform-boost-customer-experience-3605335/
  - **Marriott** deploys NoSQL for its reservation system that books $38 billion annually
    - https://diginomica.com/2015/10/07/why-marriott-is-transforming-their-legacy-systems-with-nosql/
  - **Gannett**, the No.1 U.S. newspaper publisher, uses NoSQL for its proprietary content management system, Presto
    - https://www.couchbase.com/customers/gannett
  - **GE** deploys NoSQL for its Predix platform to help manage the Industrial Internet
    - https://www.computerweekly.com/news/2240176248/GE-uses-big-data-to-power-machine-services-business

NoSQL Databases

- Unlike relational databases there are 4 different flavours of NoSQL
  - Key-value store
  - Document-based store
  - Column-based store
  - Graph-based
- Couchbase and MongoDB are some of the most popular document based databases
- MongoDB was introduced in the last lecture.

MongoDB Reminder

- Document
  - A document is a set of key-value pairs:

```
{
    studentno:      1712345,
    studentName:    “Anton Du Beke”,
    email:          “a.dubeke@wlv.ac.uk”
}
```

key

value

  - Documents have a dynamic schema, which means that documents in the same collection do not need to have the same set of columns or structure.

```
{
    studentno:      1754321,
    studentName:    “Susan Calman”,
    dob:        “11-APRIL-1999”
    email:          “s.calman@wlv.ac.uk”
}
```

DOB is new

  - Common fields in a collection's documents may hold different types of data too.

```
{
    studentno:      1754321,
    studentName:    { firstName: “Kevin”, surname: “Clifton” },
    email:          “k.clifton@wlv.ac.uk”
```

studentName now an array

Querying data

- The last lecture introduced the find() command:
  - find() can contain parameters to restrict what data is returned
    - Similar to the WHERE part of a SQL statement
  - For example, to find the Sales department details:
    - db.dept.find({dname:"SALES"}).pretty()
  - In SQL would be:

```
SELECT * FROM DEPT
WHERE dname = 'SALES';
```

find()

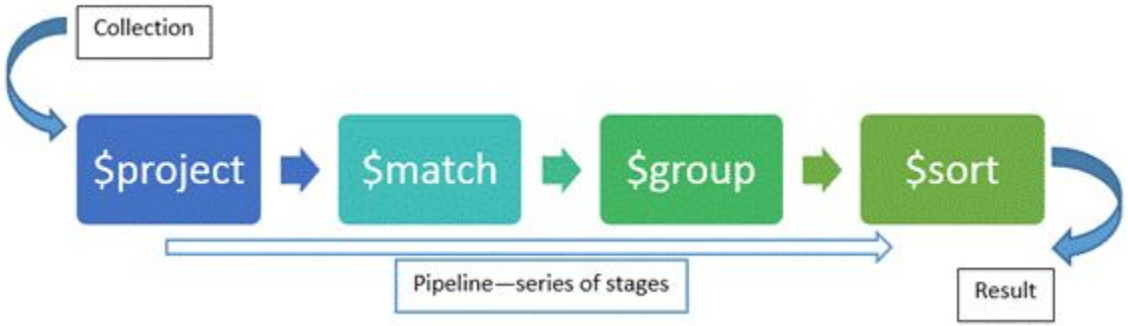
Other examples of using find:

Operation	Syntax	Example	RDBMS Equivalent
Equality	{<key>:<value>}	db.dept.find({loc:"NEW YORK"})	where loc= 'NEW YORK'
Less Than	{<key>:{$lt:<value>}}	db.dept.find({deptno:{$lt:30}})	where deptno < 30
Less Than Equals	{<key>:{$lte:<value>}}	db.dept.find({deptno:{$lte:30}})	where deptno <= 30
Greater Than	{<key>:{$gt:<value>}}	db.dept.find({deptno:{$gt:30}})	where deptno > 30
Greater Than Equals	{<key>:{$gte:<value>}}	db.dept.find({deptno:{$gte:30}})	where deptno >= 30
Not Equals	{<key>:{$ne:<value>}}	db.dept.find({deptno:{$ne:30}})	where deptno != 30

https://www.tutorialspoint.com/mongodb/mongodb_query_document.htm

Aggregation Pipeline

- More complex queries require the use of a framework called the aggregation pipeline
- MongoDB's aggregation framework is based on the concept of data processing pipelines.
  - It is similar to pipelines found in Unix/Linux.
- At the start of the process is the collection, which is searched document by document
- Documents are piped through a processing pipeline and go through a series of stages until you get a result set:

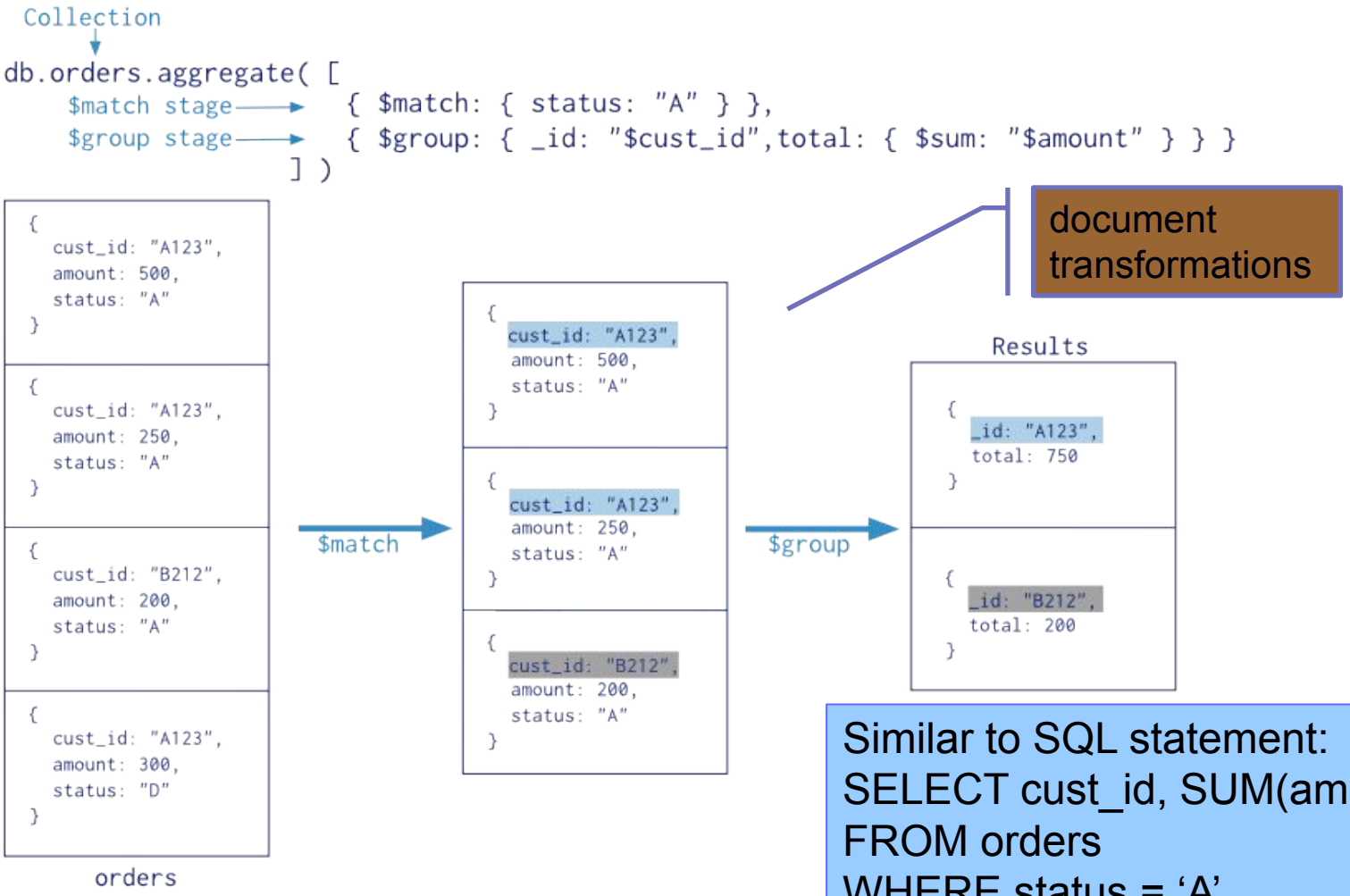


http://www.c-sharpcorner.com/article/aggregation-pipeline-in-mongodb/

Aggregation Pipeline

- Documents enter a multi-stage pipeline that transforms the documents into aggregated results.
- This is needed if you want to compute results on the data, such as sum or count
  - Similar to using GROUP BY in SQL
- Each stage in the pipeline transforms the documents as they pass through the pipeline.
- A basic pipeline provides filters that are like queries and document transformations that modify the form of the output.
- Since Version 3.2 a pipeline can be used to “join” collections.
- All the aggregation operators can be found in the MongoDB documentation:
  - https://docs.mongodb.com/manual/reference/operator/aggregation/

Pipeline Example



Similar to SQL statement:
SELECT cust_id, SUM(amount)
FROM orders
WHERE status = 'A'
GROUP BY cust_id

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

$group

- $group will take the input document and group them by a specified key and then apply the aggregate function to each group
- For example, suppose we want to sum the salaries found in the emp collection:

```
db.emp.aggregate ( [
{       $group:
  { _id: "$deptno", total: {$sum: "$sal"} } }
  ])
```

Aggregate operator

Key (column) to be processed

Key (column) we want to group by

Name for the result

Aggregate function

Similar to SQL command:
SELECT deptno, SUM(sal) AS total
FROM emp
GROUP BY deptno;

Make sure you get the brackets lined up properly!

$lookup

- $lookup can be used to provide a left outer join between two collections.
- The $lookup stage does an equality match between a column from the input documents with a column from the joined collection:

Collection providing input documents

```
db.dept.aggregate([ {
      $lookup:  {
          from: "emp",
          localField: "deptno",
          foreignField: "deptno",
          as: "employees"
        }
   }
])
```

Similar to SQL command:
*SELECT * FROM*
dept, emp
WHERE dept.deptno = emp.deptno;

Collection to join

Column to match in the dept collection (PK)

Column to match in the emp collection (FK)

Name for the output array

Other functions: count and distinct

- count can be used to count the number of documents in a collection:

  db.emp.count()
- Can also provide a query to apply a selection criteria:

  db.emp.count({deptno: 10})
- Can also add count() to a find query to count the records returned, instead of listing them:

  db.dept.find({dname:"SALES"}).count()
- distinct finds the distinct values for a specified column (similar to distinct clause in SQL):

  db.emp.distinct("deptno")

NESTED DOCUMENTS

DEPT/EMP schema

- Instead of having two separate tables, the employee details can be nested within the department:

```
db.deptCollection.insert(
{     deptno: 10,
      dname: 'ACCOUNTING',
      loc: 'NEW YORK',
      employees: [
         { empno: 7782, ename: 'CLARK',
           job: 'MANAGER',  mgr: 7839,
           hiredate: new Date('1989-06-09'),  sal: 2450
         },
         {      empno:7839, ename: 'KING',
                job: 'PRESIDENT',
                hiredate: new Date('1980-11-17'), sal: 5000
         },
         {….}
         ] } )
```

Set of employees, stored as a array

No need to store nulls, if a field does not have a value

Finding data in nested documents

- Finding data in a nested document is more complex.
- Need to use dot notation to refer to the nested fields, such as *employees.sal*

```
db. deptCollection.find(
    { "employees.sal" :
        { $gt: 2000}})
```

Will return **all** the employees in the document, even if only one employee found with this criteria!

- Employees is an array of employee types.
- $elematch() can also be used to query arrays:

```
db. deptCollection.find({deptno:10},
    { employees: {$elemMatch: {sal: { $gt: 2000}}}})
```

Will return the **first** found employee with this criteria

$filter & Nested Documents

- It is not satisfactory to return all the employees in the array if only 1 record matches
- employees is an array, which has various operators such as: **`$filter`**
  - This returns a subset of the array with only the elements that match the filter condition
  - We can use this to “gather” elements of our employees array to get the employees matching the query criteria only, rather than one, or everyone.

$filter

- **`$filter`** has the following syntax:

```
{ $filter: {
      input: <array>,            /* expression for the array */
      as: <string>,          /* variable name for the element */
      cond: <expression>     /* filter condition */
} }
```

- For example:

```
db.deptCollection.aggregate([ {
    $project: {        /* used to pass along the documents to the next stage */
      empSet: {      /* name for the aggregated set */
        $filter: {
          input: "$employees",
          as: "employee",
          cond: { $gte: [ "$$employee.sal", 2000 ] }
        }    /* filter */
      }      /* empSet */
    } /* project */
  }          /* aggregate */
])     /* Lots of brackets to match up! */
```

Object Ids

- We have seen that MongoDB creates a unique object id for objects in the database if one is not defined.
- Using `find()` will show the id generated:
  - db.projCollection.find({projno: 140}).pretty()
- The id generated can then be used to link documents together.
- For example, if the system for projno 140 is "*58245944d6473dc2e8d23a26*", to add an employee to this project:

```
db.deptCollection.insert( { deptno: 70,
    dname: 'OBJECT Test',
    loc: 'STOCKPORT',
    employees:
    [ { empno: 8199, ename: 'Perry',
    project: ObjectId("58245944d6473dc2e8d23a26")
        } ] } )
```

Pattern Matching

- When analysing data you might not want to search for an exact value
- Regular expressions can be used to search for patterns in the data
- Similar to SQL's LIKE clause
- The format for regular expressions is:
  - { <field>: /pattern/<options> }
- Or can use the $regex command:
  - { <field>: { $regex: /pattern/, $options: '<options>' } }
  - { <field>: { $regex: 'pattern', $options: '<options>' } }
  - { <field>: { $regex: /pattern/<options> } }

Pattern Matching Examples

- For example:
  - db.dept.find({dname: /SAL/})
  - db.dept.find({dname: { $regex: /RES/}})
- Use *i* option to make it case insensitive:
  - db.emp.find({ename:/sco/i})
- Nested queries:

```
db.deptCollection.find(
    {"employees.ename":/sco/i}).pretty()
```

- Downside of the above it returns all the other employees in the same department.
- One option is to use **`$unwind`**:

```
db.deptCollection.aggregate([{$unwind: '$employees'},
            {$match: {"employees.ename": /sco/i}}])
```

Indexes

- In any database system when querying large datasets indexes help improve performance
- MongoDB supports indexes with the `createIndex()` function.
- The syntax is:
  - db.collection.createindex(keys, options)
- MongoDB supports several different index types including text, geospatial, and hashed indexes:
  - https://docs.mongodb.com/manual/indexes/#index-types
- For example:
  - db.emp.createIndex({ename: "text"})

Index type

Using Indexes

- Different syntax is needed in MongoDB
- To perform a search on an index field use **`$text`**
- The syntax is:

```
{  $text:
    {
      $search: <string>,
      $language: <string>,      /* optional */
      $caseSensitive: <boolean>,
                /* defaults to false, i.e., not case sensitive */
      $diacriticSensitive: <boolean> /* defaults to false */
    }
}
```

- For example:

```
db.emp.find({$text : {$search : “scott"}})
```

Using Indexes

- Can find out what indexes are on a collection:
  db.collectionName.getIndexes()
- Indexes can be removed using the **`dropIndex()`** function

- Use the **`getIndexes()`** function to find the name of the index
- Look for the name property
  db.emp.getIndexes()
- Then use it to drop the index:
  db.emp.dropIndex("ename_text")

```
> db.emp.getIndexes()
[
        {
                "v" : 1,
                "key" : {
                        "_id" : 1
                },
                "name" : "_id_",
                "ns" : "dbcm1958.emp"
        },
        {
                "v" : 1,
                "key" : {
                        "_fts" : "text",
                        "_ftsx" : 1
                },
                "name" : "ename_text",
                "ns" : "dbcm1958.emp",
                "weights" : {
                        "ename" : 1
                },
                "default_language" : "english",
                "language_override" : "language",
                "textIndexVersion" : 3
        }
]
```

External *Found* Data

- For this module we are concerned with manipulating *found* data rather than creating new MongoDB documents
- Typical formats for export are CSV, JSON or XML
  - For example, Twitter data can be exported in JSON format, which can be manipulated using MongoDB.
- XML as a data transfer format is becoming less popular
- JSON seen as more light-weight
  - Less verbose
  - *"The Fat-Free Alternative to XML"* http://json.org/xml.html
- XML uses more words than necessary
- XML structure is not intuitive, making it hard to represent in code.

XML Data

- Some data is still in this format, so what to do with it?
  - Use Native XML database (NXD)
  - Use a XML enabled database, which is typically a relational database that offers XML support, e.g., Oracle
- NXD database are less popular
- XML enabled databases usually offer the following support for storing XML data:
  - Store XML in a CLOB (character large object)
  - Shred XML into a series of tables
  - Store XML in native XML

XML V's JSON

- Many webpages recommend using JSON over XML:
  - https://www.quora.com/What-are-the-advantages-of-JSON-over-XML
  - https://www.programmableweb.com/news/xml-vs.-json-primer/how-to/2013/11/07
  - https://www.sitepoint.com/json-vs-xml/
  - https://www.w3schools.com/js/js_json_xml.asp
- Consensus is it is seen as being simpler
- We will concentrate on JSON in this module

JSON and Relational databases

- Increasingly relational databases are offering support to handle JSON data.
- For example, Oracle 12c V2 allows JSON columns:

```
CREATE TABLE dept_json(
      deptno NUMBER(2) NOT NULL,
      dname VARCHAR2(15),
      emp_doc CLOB CONSTRAINT ensure_json
            CHECK (emp_doc IS JSON));
```

- Inserts:

```
INSERT INTO dept_json
  VALUES (10, 'ACCOUNTING',
        '{ "empno"          : 7782,
          "ename"           : "CLARK",
          "contactDetails"  : {"phoneExt" : 1483, etc...}');
```

- Querying nested values using dot notation:

```
SELECT d.emp_doc.contactDetails.phoneExt
FROM dept_json d;
```

Twitter Data

- Twitter data is in JSON format
- MongoDB can be used to load JSON data
- Need to use the `mongoimport` command
- Format is:

```
mongoimport
 --db dbYourStudentNo
 --username yourStudentNo
 --password yourPassword
 --file fileToBeLoaded.json
 --collection collectionName
```

Twitter Data

- For example, some sample Weather data can be found on Canvas in a file called `weather.json`
- To import this assuming a student number of 0123456:

```
mongoimport
 --db db0123456
 --username 0123456
 --password my0123456Password
 --file weather.json
 --collection weather
```

- Note the above command should be entered at the OS command line, not within Mongo
- It should also be entered on one line only

Twitter Data

- NoSQL databases are referred to as Schema-less databases
- Means there are no handy Data Dictionary tables to query what columns are in our database
- It does not mean there is no structure at all, just that there is not a constant structure found in all documents
- Means you need to do some analysis of the data to find out what sort of information it contains
- Remember the Analysis stage from Lecture 2
- To see some data:

  db.weather.find()
- It will not show all the records!
  - You need to type in "it" for more
- To see how many records there are:

  db.weather.count()

Twitter Data - Pattern matching

- As mentioned MongoDB uses Regular Expressions to do pattern matching
- For example, find any documents where `snow` has been mentioned in the *text* field:

```
db.weather.find({text: /snow/}, {text:1, _id:0})
```

- Or starting with `snow`:

`i` will make the search case-insensitive

```
db.tweets.find({content: /^snow/i}, {text:1, _id:0})
```

- Or ending with `snow`:

```
db.tweets.find({content: /snow$/i}, {text:1, _id:0})
```

- Note, the second part of the find command:

```
{text:1, _id:0}
```

- means we will only show the text field.
  - _id: 0 will suppress the id field

Cleaning the Data

- As with any external data set the information found can contain rubbish data.
- The tweets in Twitter data in particular can be subject to:
  - Abbreviations
  - Swear words!
  - Misspelling
  - Unnecessary data
- You may want to reshape the data to:
  - just include what is wanted
  - produce a consistent view of the data to make any statistical operations easier.
- See the second MongoDB workbook for details on how to do this.

Python Notebook

- On Canvas there is also a Python Notebook.
- Use of this is optional
  - Has the advantage of viewing the twitter data more easily
  - Can also add your own comments in a text field and makes it easier to save the commands and results
- Note:
  - The syntax in the Mongo Shell and Python Notebook is slightly different.
  - For example, instead of:

    db.weather.distinct("lang")
  - In Python you can store a reference to the collection, e.g., `wcol` and use it instead:

    wcol.distinct("lang")
  - Remember to change the login details at the start to your own username and MongoDB password
  - Some commands only work in one of the environments
    - E.g., you must use `mongoimport` to load the data on mi-linux.
  - The comments will give further information

Conclusion

- Big data is a big research area currently
  - The data generated will not be going away anytime soon, so need an effective way of handling it
- NoSQL
  - Lots of different types of projects.
  - Many of the examples listed are open-source.
  - RDBMS could evolve to provide better support for non-structured data.
- *Horses for courses*
  - Need to appreciate the strengths and weaknesses of each type of database, so you can pick the most appropriate tool for the data being stored