NoSQL Databases Part 2

NoSQL

MongoDB – Further querying

NoSQL verses relational DBMS

Why NoSQL?

- Modern applications cover and include the web, mobile and Internet of Things and have different characteristics to traditional enterprise applications, such as HR and ERP.
- They need to (Couchbase, n.d.):
  - Support large numbers of concurrent users (tens of thousands, perhaps millions)
  - Deliver highly responsive experiences to a globally distributed base of users
  - Be always available – no downtime
  - Handle semi- and unstructured data
  - Rapidly adapt to changing requirements with frequent updates and new features
- Building such systems have a new technology requirements.
- New enterprise technology architecture needs to be agile and accommodate unprecedented levels of scale, speed and data variability
- Companies are tuning to NoSQL database technology to achieve this

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()

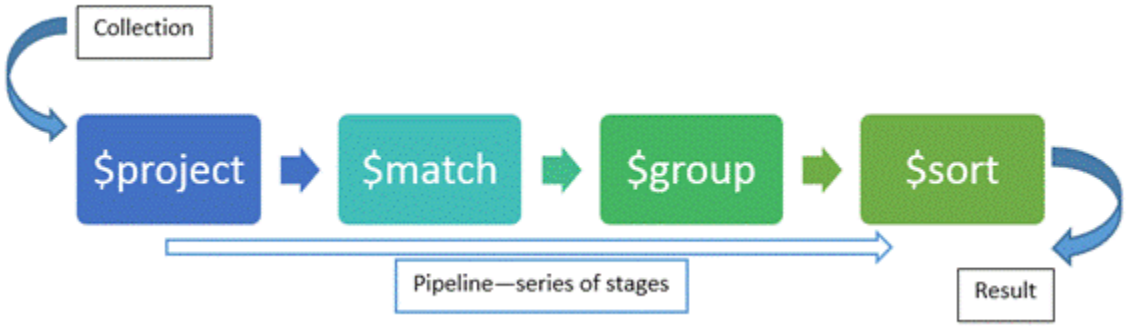
Other examples of using find:

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:

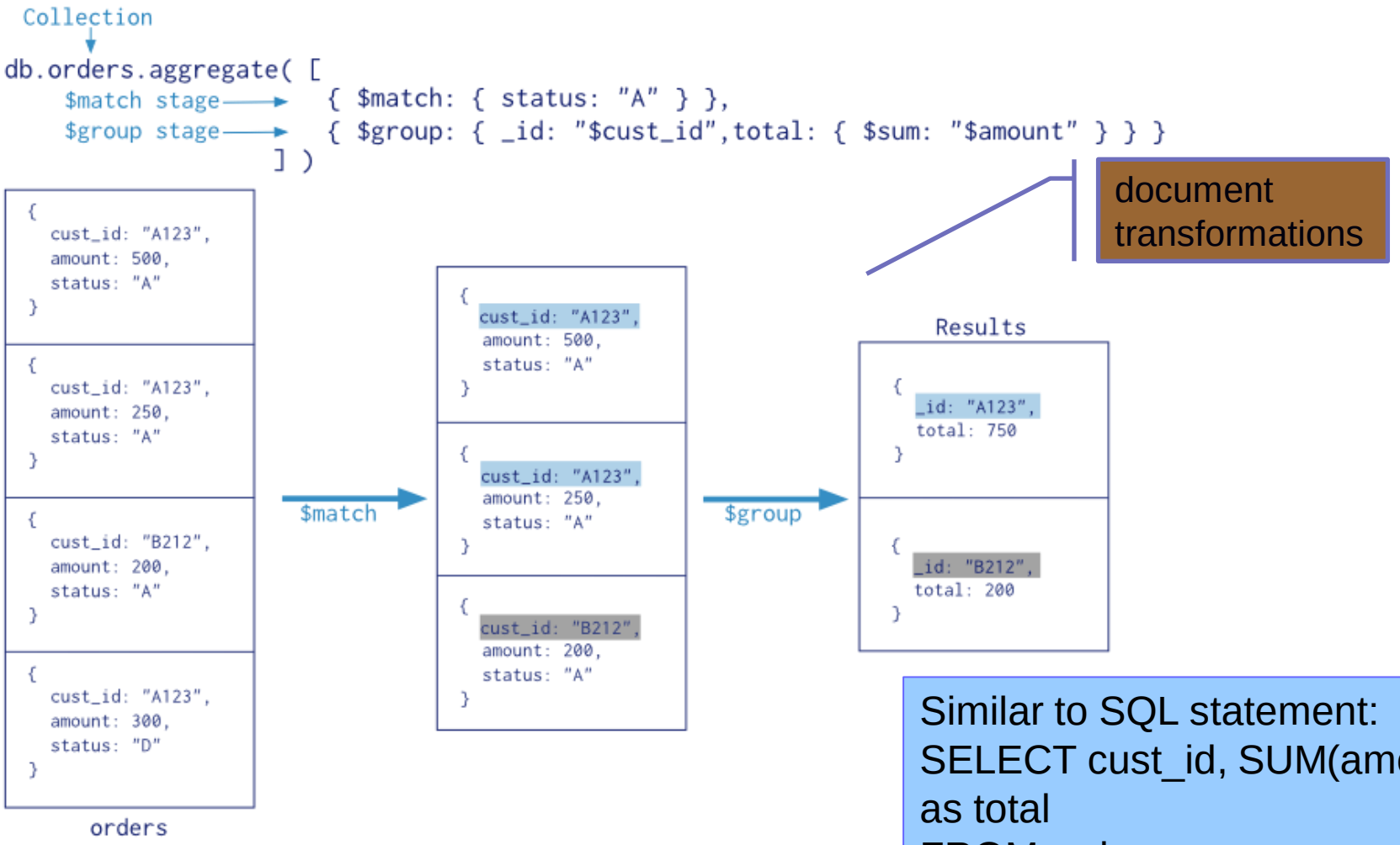


http://www.c-sharpcorner.com/article/aggregation-pipeline-in-mongodb/

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) as total
FROM orders
WHERE status = 'A'
GROUP BY cust_id

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

$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

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

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

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

- Using indexes in a relational database does not change how you query the system.
- 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
- For example, Twitter data is in JSON format, which can be manipulated using MongoDB.
- Many webpages that advocate 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

JSON and Relational databases

- A lot of systems will now use JSON format for data exchange, so 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;
```

NOSQL V'S RELATIONAL

NoSQL V's Relational

- Does these NoSQL systems mean the end of relational DBMS?
- No quite, relational DBMS are still appropriate for structured data.
- However, we are entering an age of "polyglot persistence"
  - This is a technique that uses different data storage technologies to handle varying data storage needs.
  - Polyglot persistence can apply across an enterprise or within a single application.

Why NoSQL Will Not Kill the RDBMS

- There seems to be three major focuses of the NoSQL movement (Mohawksoft.org):
  - scalability of storage
    - All systems (including NoSQL) will hit limitations when you increase the load
    - The strategy of NoSQL is to create a system that can easily grow beyond any “current” implementation should the need arise.
  - rigid schemas
    - Most data does not fit easily into a normalised schema
  - non-scalability of relational data access
    - “joins don't scale”
    - Joining tables will be less efficient than reading from a data store

Why NoSQL Will Not Kill the RDBMS – Rebuttal: Scalability

- "Scalability" is a difficult problem to quantify - what should be scaled?
- The big problem:
  - quantifying what you want to scale:
    - data storage, like a remote backup service?
    - small transactions like twitter?
    - mostly page views like Google?
  - What is your ratio between reads and writes?
  - What kind of information does your site serve?
  - How coherent must your data be? Do you need to guarantee data integrity?
- To have any meaningful discussion on scalability, you need to quantify what it is you wish to scale and the limitations imposed by requirements.
- Don't forget RDBMS databases can be scaled with faster hardware, data partitioning, better caching, and etc.

Why NoSQL Will Not Kill the RDBMS – Rebuttal: Rigid Schemas

- Even in the most *"schemaless"* database, there is structure in the way the data is stored
  - How would the system access it otherwise?
  - If we accept that there must be some order and control over the data storage, whether or not we call it a schema is unimportant.
- SQL "schemas" are not inflexible
  - One can add and usually remove columns in a SQL table.
- Could create "generic" key/value tables in a standard SQL database where the "value" column contains any sort of data.
- In NoSQL solutions which do not enforce a data definition, you can not search by data contained in "value," so how is the SQL implementation any different?
  - In addition, many SQL databases allow you to create an index based on the results of a function.
  - If the "value" in your key/value table contains something like XML or JSON, you can create an index from specific member values extracted from a parser function.

Why NoSQL Will Not Kill the RDBMS – Rebuttal: Joins

- A SQL RDBMS is a comprehensive set of tools, it does not require that data be 100% normalized.
- Just because an RDBMS supports joins, triggers, foreign key constraints, primary keys and so on, does not mean that you need to use them.
- You design your system for your needs.
  - If you don't want a join, put all your data in one table.
  - You want a key/value store? Create a key/value table:

```
CREATE TABLE myKeyValue (
    key         NUMBER(5),
    Value       XMLType );  /* or could be BLOB, or CLOB */
```

- You still get all the tools and features of the RDBMS if you need them.
- In a NoSQL system, you don't have that option.

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.
  - The fact there is no well-defined definition could lead to its downfall.
  - 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