Big Data Management

F21BD

Assignment One Report – NoSQL Data Storage

Heba El-Shimy

H00280277

Data Science MSc. Program

2017 - 2018

Table of Contents

Abstract

This report is a detailed walkthrough of the steps taken to migrate a dataset from a RDBMS (SQL) to a NoSQL DBMS.

The dataset provided was a modified version from Open Flights dataset. It consists of 6,162 airlines, 7,184 airports, connected by 67,663 routes. There are also 6,649 cities located in 260 countries.

The intended use for the dataset is building a flight comparison website, where information about flights can be queried easily and with minimum response time and allow for flexibility of modifications that may be applied later to the dataset.

The report discusses the current solution's drawbacks, reasons for migration, the process of choosing a suitable DBMS system and a demonstration for that system in action.

Task 1

Brief Overview of chosen DBMS

Current Solution

The dataset is currently stored in a *Relational Database*, specifically a *MySQL* database.

Dataset used and why it can be categorized as Big Data

Big Data is a term used not only to define datasets that contain massive amounts of data (large *Volume*), but also datasets in which data can be characterized by:

- *Variety:* The type and nature of the data, whether structured (as text) or unstructured (as multimedia).
- *Velocity:* The speed at which the data is generated and processed.
- *Veracity:* The quality of the captured data (inconsistent, incomplete, deceitful)

The Flights dataset currently in use falls under the category of *Big Data* as it is in *high volume* (tens of thousands of records) and has some *veracity* as the data may be inconsistent or incomplete.

Handling *Big Data*

As discussed above, handling *Big Data* requires specific measures to be taken regarding data storage. One of the most important requirements is having *Distributed systems* with many nodes across multiple servers that have *replicas* and/or *shards* of the data. This approach ensures the application can handle high concurrency of interactions and queries by load balancing across all nodes that have the same set of data (*replicas*), as well as being fault tolerant if one or more of the nodes is offline, then traffic gets redirected to other nodes temporarily.

Using NoSQL approach (MongoDB)

I decided upon using *MongoDB* as a *NoSQL* database for migrating the dataset from the current *MySQL* database.

The requirements imposed by our use case in the *flight comparison* website, is having a storage system that can:

- Handle *high volumes* of data,
- Withstand *increased loads of traffic* with *high availability* and *fault tolerance*,
- Handle millions of *read/write* operations while providing *quick responses*,
- *Scale* easily and
- *Replicated* across multiple servers/datacenters.

My decision of using *MongoDB* specifically for this use case was based on the fact that the dataset is rich with hierarchical structure and that the *flights comparison* website will need more frequent *read* than *write* operations.

SQL vs. NoSQL Systems

	SQL Systems (Relational Database Model)	NoSQL Systems (MongoDB)
Storage Paradigm	- Data is organized into one or more **tables** of columns and rows, with a **unique key** identifying each row. The rows represent instances of that type of entity and the columns represent values attributed to that instance.	- ***MongoDB*** is a ***Document Database***. It keeps all information related to a single entity in one **document** contrary to ***Relational databases*** where multiple tables are created to model one-to-many relationships. The complete data of a document can be stored and retrieved atomically. A **document** is an alternate to a row while a **collection** holds multiple documents and is the alternate to a table in SQL.
	- Data lookup is done by queries (in SQL) selecting whole rows or selected attributes from specific rows, ***indexing*** is supported to speed up the data retrieval process.	- Data lookup is done by queries selecting documents or selected fields from specific documents, ***indexing*** is supported to speed up the data retrieval process.
	- The ***Relational Database*** Model allows for creating **relationships** between different entities in different tables, using ***Foreign Keys***. This approach is used extensively if the database is to be ***Normalized***. Database ***Normalization*** basically involves splitting related data into different tables and have relationships between these tables. The ***Normalization*** of a database helps minimize data redundancy.	Modeling data in ***Document Databases*** usually makes use of ***denormalization*** as a technique of storing the data.

	SQL Systems (Relational Database Model)	NoSQL Systems (MongoDB)
	- Data must conform to a pre-defined ***schema*** which defines the structure of all the tables and relationships between them. This ***schema*** cannot be easily changed afterwards, so it is fixed.	- ***Document Databases*** store data in formats such as XML or JSON, short for JavaScript Object Notation, which can hold any kind of hierarchically structured data. No need for a pre-defined ***schema***.
		- ***Document Databases*** can be considered ***Key/Value Stores*** with extra capabilities to structure and query the data.

	SQL Systems (Relational Database Model)	NoSQL Systems (MongoDB)
CAP Theorem (*Consistency, Availability and Partition Tolerance)* Distributed systems can only guarantee **two** of the features, not all three.	Highly ***consistent*** systems due to many constraints to preserve data the same across all nodes, this comes at the cost of availability. *ACID* properties: • **Atomicity:** an indivisible series of database operations such that either all occur, or nothing occurs. This is different from the (*A*) in the definition of *CAP Theorem*, where in the latter it refers to *Availability* of the data. • **Consistency:** transactions preserve all the database protocols and rules. In contrast, the (*C*) in *CAP* refers only to single-copy consistency. • **Isolation:** concurrent execution of transactions results in a system state that would be obtained if transactions were executed sequentially, so each transaction is independent unto itself. • **Durability:** ensures that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.	Easily scalable systems, so are most of them are basically ***available*** and ***partition tolerant*** but at the price of ***consistency*** where they become ***eventually consistent***. *BASE* properties: • **Basically Available:** the system does guarantee the availability of the data as regards *CAP Theorem*; there will be a response to any request. But, that response could still be 'failure' to obtain the requested data or the data may be in an inconsistent or changing state. • **Soft state:** replicated and cached data. • **Eventual consistency:** the system will eventually become consistent once it stops receiving input. In our case, ***MongoDB***, is considered (**CP**), i.e., ***consistent*** and ***partition tolerant*** at the cost of availability.

Benefits of *MongoDB* over *MySQL* for our use case

	MySQL (Current Database)	MongoDB (Proposed Solution)
Performance (Loads on the system due to the amount of interactions and devices using it concurrently)	- Heavy workloads hit the limits for *RDBMS* models performance resulting in slow response times.	- Low latency reads and updates, real-time performance without compromising the robustness of the system.
	- Retrieving data from *RDBMS* models involves multiple join operations to gather the required data from different tables, which is computationally expensive and slows performance.	- No *join* operations as all related data are stored in one document, hence, *read* and *write* operations are computationally cheap and fast.
	- Data stored in *RDBMS* are subjected to many constraints during writing operations, to ensure consistency of data specially across distributed systems, this adds another factor that slows performance.	- No constraints on data to comply to a specific schema or set of rules, so *write* operations happen much faster.
Flexibility	- Rigid systems due to pre-defined schemas.	- Flexible, schema-less. Easily change document structures or introduce unstructured data.
Scalability	1. *Scale Vertically (scale-up)*: Most common solution, but eventually limited as it requires getting new hardware with more storage capacity and processing power. 2. *Scale Horizontally (scale-out)*: Use of data replication and sharing to obtain a *distributed system* which is a costly and complex process.	1. *Scale Vertically (scale-up)* 2. *Scale Horizontally (scale-out)* Both solutions are easily achievable; we will rely mainly on scaling-out by adding more nodes having replicas of the dataset.

	MySQL (Current Database)	MongoDB (Proposed Solution)
Fault Tolerance	- Low fault tolerance due to difficulty of obtaining highly available *distributed systems*.	- High fault tolerance due to ease of scaling-out and having multiple nodes.
Maintenance (Keeping systems running smoothly, adding nodes to scale, keeping processes up and debugging components that could go wrong)	- Difficult to maintain, specially if within a *distributed system*. This is mainly due to the complexity added during implementation to have tables and relationships properly representing the data while being replicated and partitioned among multiple servers.	- Easily maintainable as the system relies on components that have simple mechanisms and architecture underlying them.

Task 2

Discussion on the Data Model of Choice

While examining the current data model, and based on the relationships between entities show in Figure 1 below, I found that data is *normalized*, related data is separated between different tables while having relationships between them using *foreign keys*. This approach is the essence of *Relational Databases* as it reduces redundancy.

Migrating to *MongoDB*, *normalization* is not a good approach because it does not allow *join* operations. Hence, I tried to use a *denormalized* approach when creating an alternate model for the data in *MongoDB*. I achieved that by using the concept of *embedding*, where I embedded related data as sub-documents into other documents to create a hierarchical structure. At other places, I only *embedded* an *ObjectID* referencing the main document.

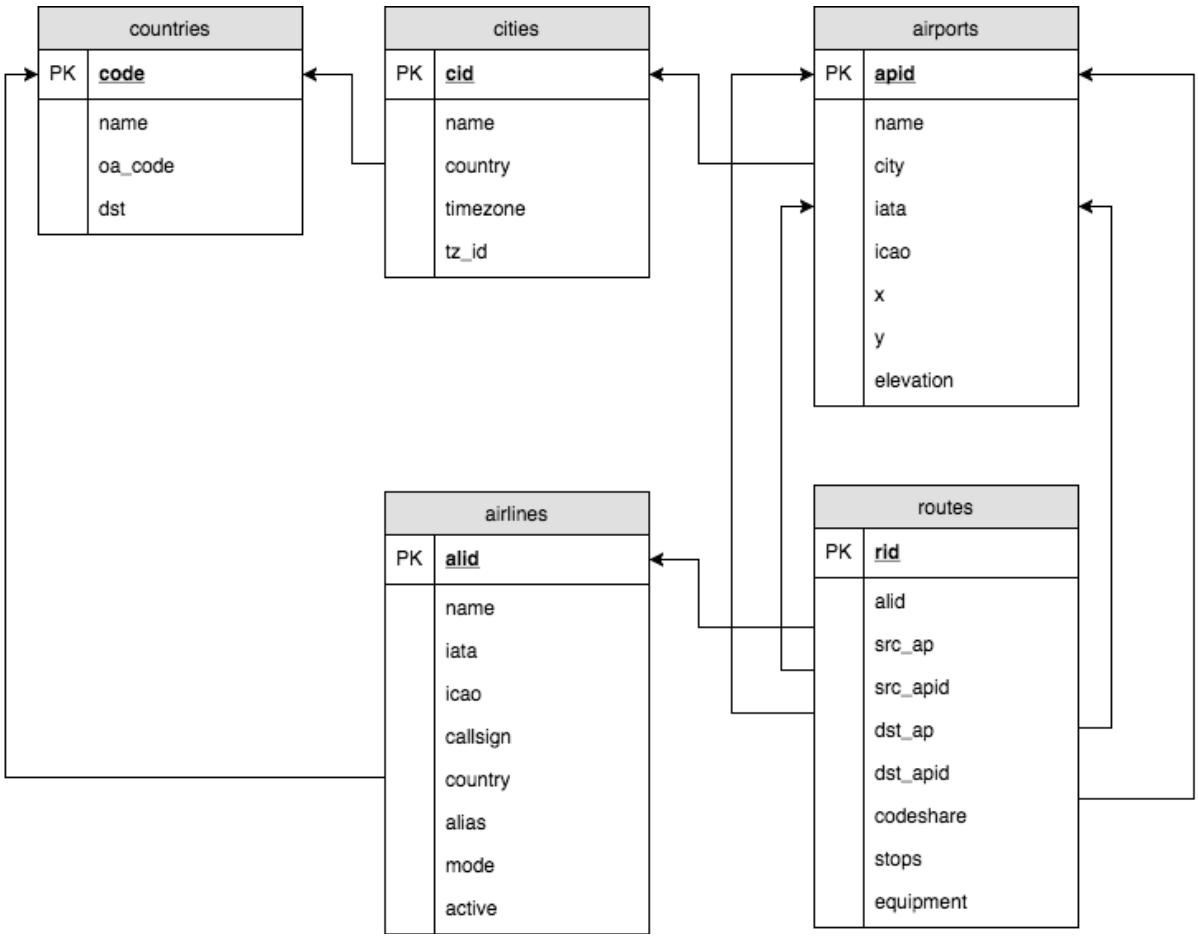


Figure 1: Open Flights Dataset as represented in a RDBMS

The justification behind this approach is to somewhat have a balance between:

- Flexibility (having separate collections that can be updated and can grow freely). *MongoDB* has a limit of 16MB for the size of a single document, adding entire collections as subdocuments can exceed this document limit and at the same time can be a hassle to update or grow.

- Fast response in data retrieval in the least amount of round trips for data that will most probably be queried together.

Converting Countries Table into Collection

Countries table has 2 relationships:

- One-to-Many Relationship with **Cities** table, where a country can have one or more record in the **cities** table. The **primary key** in the **Countries** table is the country **code**, which is used at the same time as a **foreign key** to reference the country inside the **cities** table (creating a relationship).

- Many-to-Many Relationship with **Airlines** table, where one or more records in **Countries** table can have one or more record in the airlines table. The **primary key** in the **Countries** table is the country **code**, which is used at the same time as a **foreign key** to reference the country inside the airlines table (creating a relationship).

When migrating to *MongoDB*, I copied both the **cities** and the **airlines** tables into an array embedded inside the parent **countries** collection. I did not copy all the fields in **cities** and **airlines**, I only copied and embedded the fields that would be most queried together (airline name and iata and city name and timezone id), and embedded an **ObjectID** to reference a parent **Cities** collection and parent **Airline** collection.

Countries Collection JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "array",
  "items": {
      "$ref": "#/definitions/Country"
  },
  "definitions": {
      "ID": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "$oid": {
                  "type": "string"
              }
          },
          "required": [
              "$oid"
          ],
          "title": "_id"
      },
      "Airline": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "name": {
                  "type": "string"
              },
              "iata": {
                  "type": "string"
              },
              "alid": {
                  "$ref": "#/definitions/ID"
              }
          },
          "required": [
              "alid",
              "iata",
              "name"
          ],
          "title": "airline"
      },
      "City": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "cid": {
                  "$ref": "#/definitions/ID"
              },
              "name": {
                  "type": "string"
              },
              "tz_id": {
                  "type": "string"
              }
          },
          "required": [
              "cid",
              "name",
              "tz_id"
```

```
            ],
            "title": "city"
        },
        "Country": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "_id": {
                    "$ref": "#/definitions/ID"
                },
                "name": {
                    "type": "string"
                },
                "oa_code": {
                    "type": "string"
                },
                "dst": {
                    "type": "string"
                },
                "airlines": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/Airline"
                    }
                },
                "cities": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/City"
                    }
                }
            },
            "required": [
                "_id",
                "airlines",
                "cities",
                "dst",
                "name",
                "oa_code"
            ],
            "title": "Country"
        }
    }
}
```

Converting Cities Table into Collection

Cities table has 2 relationships:

• One-to-One Relationship with **Countries** table, where a city can belong only to one record in the **countries** table. The **country** column values are references to the related country in the **countries** table.

• One-to-Many Relationship with **Airports** table, where a city can have one or more record in the airports table. The **primary key** in the **Cities** table is the city **id**, which is used at the same time as a **foreign key** to reference the city inside the airports table (creating a relationship).

I copied **airports** table into an array embedded inside the parent **cities** collection. I did not copy all the fields in **airports**, I only copied and embedded the following fields (airport name and iata), and embedded an **ObjectID** to reference a parent **Airports** collection.

Cities Collection JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "array",
  "items": {
      "$ref": "#/definitions/City"
  },
  "definitions": {
      "ID": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "$oid": {
                  "type": "string"
              }
          },
          "required": [
              "$oid"
          ],
          "title": "_id"
      },
      "Airport": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "name": {
                  "type": "string"
              },
              "iata": {
                  "type": "string"
              },
              "apid": {
                  "$ref": "#/definitions/ID"
              }
          },
          "required": [
              "apid",
              "iata",
              "name"
          ],
          "title": "airport"
      },
      "City": {
```

```
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "_id": {
                    "$ref": "#/definitions/ID"
                },
                "name": {
                    "type": "string"
                },
                "country": {
                    "$ref": "#/definitions/ID"
                },
                "timezone": {
                    "type": "null"
                },
                "tz_id": {
                    "type": "null"
                },
                "airports": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/Airport"
                    }
                }
            },
            "required": [
                "_id",
                "airports",
                "country",
                "name",
                "timezone",
                "tz_id"
            ],
            "title": "City"
        }
    }
}
```

Converting Airports Table into Collection

Airports table has many relationships:

• One-to-One Relationship with **Cities** table, where an airport can belong only to one record in the **cities** table. The **city** column values are references to the related country in the **countries** table.

• Many-to-One Relationship with **Routes** table, where more than one airport record in **Airports** table can have one related record in the **Routes** table. Foreign Keys (src_ap,

src_apid, dst_ap, dst_apid) in **Routes** table are used to reference the following columns in **Airports** table respectively (iata, apid—primary key).

I embedded selected fields from **Airports** into **Cities** collection while keeping a separate **Airports** collection with all fields copied from the *MySQL* table.

Airports Collection JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "array",
  "items": {
      "$ref": "#/definitions/Airport"
  },
  "definitions": {
      "ID": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "$oid": {
                  "type": "string"
              }
          },
          "required": [
              "$oid"
          ],
          "title": "_id"
      },
      "Airport": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "_id": {
                  "$ref": "#/definitions/ID"
              },
              "name": {
                  "type": "string"
              },
              "city": {
                  "$ref": "#/definitions/ID"
              },
              "icao": {
                  "type": "string"
              },
              "x": {
                  "type": "number"
              },
              "y": {
                  "type": "number"
              },
              "elevation": {
                  "type": "integer"
              }
          },
          "required": [
              "_id",
              "city",
              "elevation",
              "icao",
```

```
                "name",
                "x",
                "y"
            ],
            "title": "Airport"
        }
    }
}
```

Converting Routes Table into Collection

Routes table has many relationships:

• Many-to-One Relationship with **Airlines** table, where one or more routes can belong to one record in the **airlines** table. The **alid** column values are references to the related airline in the **airlines** table.

• One-to-Many Relationship with **Airports** table, where one record in **Routes** table can have one or more related record in the **Airports** table. Foreign Keys (src_ap, src_apid, dst_ap, dst_apid) in **Routes** table are used to reference the following columns in **Airports** table respectively (iata, apid—primary key).

I embedded selected fields from **Routes** into **Airlines** collection while keeping a separate **Routes** collection with all fields copied from the *MySQL* table.

Routes Collection JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "array",
  "items": {
      "$ref": "#/definitions/Route"
  },
  "definitions": {
      "ID": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "$oid": {
                  "type": "string"
              }
          },
          "required": [
              "$oid"
          ],
          "title": "_id"
      },
```

```
      "Route": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "_id": {
                  "$ref": "#/definitions/ID"
              },
              "alid": {
                  "$ref": "#/definitions/ID"
              },
              "src_ap": {
                  "type": "string"
              },
              "src_apid": {
                  "$ref": "#/definitions/ID"
              },
              "dst_ap": {
                  "type": "string"
              },
              "dst_apid": {
                  "$ref": "#/definitions/ID"
              },
              "codeshare": {
                  "type": "string"
              },
              "stops": {
                  "type": "string"
              },
              "equipment": {
                  "type": "string"
              }
          },
          "required": [
              "_id",
              "alid",
              "codeshare",
              "dst_ap",
              "dst_apid",
              "equipment",
              "src_ap",
              "src_apid",
              "stops"
          ],
          "title": "Route"
      }
  }
}
```

Converting Airlines Table into Collection

Airlines table has many relationships:

• One-to-Many Relationship with **Routes** table, where one airline can have one or more records in the **Routes** table. The **alid** column values are references to the related airline in the **Routes** table.

• Many-to-Many Relationship with **Countries** table, where one or more record in **Airlines** table can have one or more related record in the **Countries** table. Foreign Keys (country) in **Airlines** table are used to reference the (code—primary key) column in **Countries**.

I embedded selected fields from **Airlines** into **Countries** collection while keeping a separate **Airlines** collection with all fields copied from the *MySQL* table.

Airlines Collection JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "array",
  "items": {
      "$ref": "#/definitions/Airline"
  },
  "definitions": {
      "ID": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "$oid": {
                  "type": "string"
              }
          },
          "required": [
              "$oid"
          ],
          "title": "_id"
      },
      "Route": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
              "src_ap": {
                  "type": "string"
              },
              "dst_ap": {
                  "type": "string"
              },
              "stops": {
                  "type": "string"
              },
              "rid": {
                  "$ref": "#/definitions/ID"
              }
```

```
            },
            "required": [
                "dst_ap",
                "rid",
                "src_ap",
                "stops"
            ],
            "title": "route"
        },
        "Airline": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
                "_id": {
                    "$ref": "#/definitions/ID"
                },
                "name": {
                    "type": "string"
                },
                "iata": {
                    "type": "string"
                },
                "icao": {
                    "type": "string"
                },
                "callsign": {
                    "type": "string"
                },
                "country": {
                    "$ref": "#/definitions/ID"
                },
                "alias": {
                    "type": "string"
                },
                "mode": {
                    "type": "string"
                },
                "active": {
                    "type": "string"
                },
                "routes": {
                    "type": "array",
                    "items": {
                        "$ref": "#/definitions/Route"
                    }
                }
            },
            "required": [
                "_id",
                "active",
                "alias",
                "callsign",
                "country",
                "iata",
                "icao",
                "mode",
                "name"
            ],
            "title": "Airline"
        }
    }
}
```

Task 3

Comparison between different NoSQL Systems and their suitability for our use case

	Key-Value DBs	Document DBs	Wide Column DBs	Graph DBs
Data Storage	Maps keys (an identifier) to values (implemented as an opaque binary object decoded by DB application)	Data stored in *documents* in a format such as XML or JSON (self-describing formats because they include descriptive labels on what data is stored)	Similar to the *Relational Model* in that they store data in tables—but they are much more flexible, by allowing adding new columns on the fly.	Use a branch of mathematics known as graph theory, which represents entities as "vertexes" connected by "edges." The edges show relationships between entities.
Examples	Redis, Memcached	MongoDB, CouchDB	Cassandra, HBase	Neo4j, OrientDB
Strength Points	- Speed when reading and writing data. - Quick responses even at peak time. - The ability to scale well.	- Data is in a hierarchical structure. - Modifying and retrieving hierarchical records are cheap operations. - Best for quick *read* operations.	- Massive collection of little-structured data. - Quick response times when analyzing masses of data.	- Focuses on relationships between entities. - Ability to traverse the relationships of a large number of entities efficiently.

	Key-Value DBs	Document DBs	Wide Column DBs	Graph DBs
Weaknesses	- Data integrity and the ability to comfortably query data is compromised.	- Cannot model many-to-many relationships. - Less suitable for applications that collect huge masses of data.	- Values of a dataset are spread among several columns or even several machines. - Reading complete datasets can be more time-consuming.	- Less suitable for searching sets of nodes by their properties. - Difficult to partition a graph of data among several servers to support clustering.
Use cases	Store data that is not critical to the application, but that is in high volumes and velocity. Example: status information of an online game.	Generalized use.	Analytical operations (calculations, aggregates and minimum or maximum values). Example: generating reports on business data.	Keeping track of who of your users knows whom and showing common friends of any two users. Example: social networks.
Suitability for our use case (Scale of 1-5, as all options can be suitable but at different levels)	2	5	4	3

Our *Flights Comparison* website needs a database that can provide: **data integrity**, **quick read responses**, **easily query and search for entities using their properties**, **easily scalable**.

From the above requirements imposed by our use case and from the comparison table, it can be said that *Document Databases* is a good option for migration.

Task 4

Explanation of ETL Pipeline used

I used a command line tool called *Mongify* that's built in Ruby to translate *MySQL* tables into *MongoDB* collections. I did the following steps:

Step 1: Importing SQL Dump into a MySQL Database

- I started by creating a new MySQL Database and gave it the name of *flights_database*. I also created a new user and grant it all privileges for that host.

```
Heba@MacBook Pro:~/Documents/Big Data Course/CW1$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.21 Homebrew

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>

mysql> CREATE USER 'heba'@'localhost' IDENTIFIED BY 'abeh';
Query OK, 0 rows affected (0.01 sec)

mysql> GRANT ALL PRIVILEGES ON * . * TO 'heba'@'localhost';
Query OK, 0 rows affected (0.01 sec)

mysql> FLUSH PRIVILEGES;
Query OK, 0 rows affected (0.00 sec)

mysql>

mysql> CREATE DATABASE flights_database;
Query OK, 1 row affected (0.00 sec)

mysql>
```

```
+--------------------+
| information_schema |
| flights_database   |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql> USE flights_database;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql>
```

- Importing data from sql dump file into *flights_database* using the command line.

```
Heba@MacBook Pro:~/Documents/Big Data Course/CW1$ mysql -u heba -p flights_database < flight-data-dump.sql
Enter password:
```

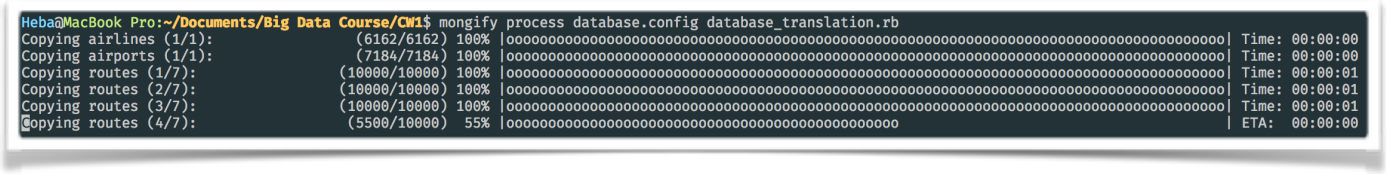
```
mysql> SHOW TABLES;
+-----------------------------+
| Tables_in_flights_database |
+-----------------------------+
| airlines                    |
| airports                    |
| cities                      |
| countries                   |
| routes                      |
+-----------------------------+
5 rows in set (0.00 sec)

mysql> DESCRIBE countries;
+---------+------------+------+-----+---------+-------+
| Field   | Type       | Null | Key | Default | Extra |
+---------+------------+------+-----+---------+-------+
| code    | varchar(2) | NO   | PRI | NULL    |       |
| name    | text       | YES  |     | NULL    |       |
| oa_code | varchar(2) | YES  |     | NULL    |       |
| dst     | char(1)    | YES  |     | NULL    |       |
+---------+------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

| Field    | Type       | Null | Key | Default | Extra          |
+----------+------------+------+-----+---------+----------------+
| name     | text       | YES  |     | NULL    |                |
| iata     | varchar(2) | YES  | MUL | NULL    |                |
| icao     | varchar(3) | YES  | MUL | NULL    |                |
| callsign | text       | YES  |     | NULL    |                |
| country  | char(2)    | YES  | MUL | NULL    |                |
| alid     | int(11)    | NO   | PRI | NULL    | auto_increment |
| alias    | text       | YES  |     | NULL    |                |
| mode     | char(1)    | YES  |     | F       |                |
| active   | varchar(1) | YES  |     | N       |                |
+----------+------------+------+-----+---------+----------------+
9 rows in set (0.00 sec)
```

Step 2: Extraction + Transformation + Loading (Using *Mongify* to translate SQL tables into MongoDB Collections)

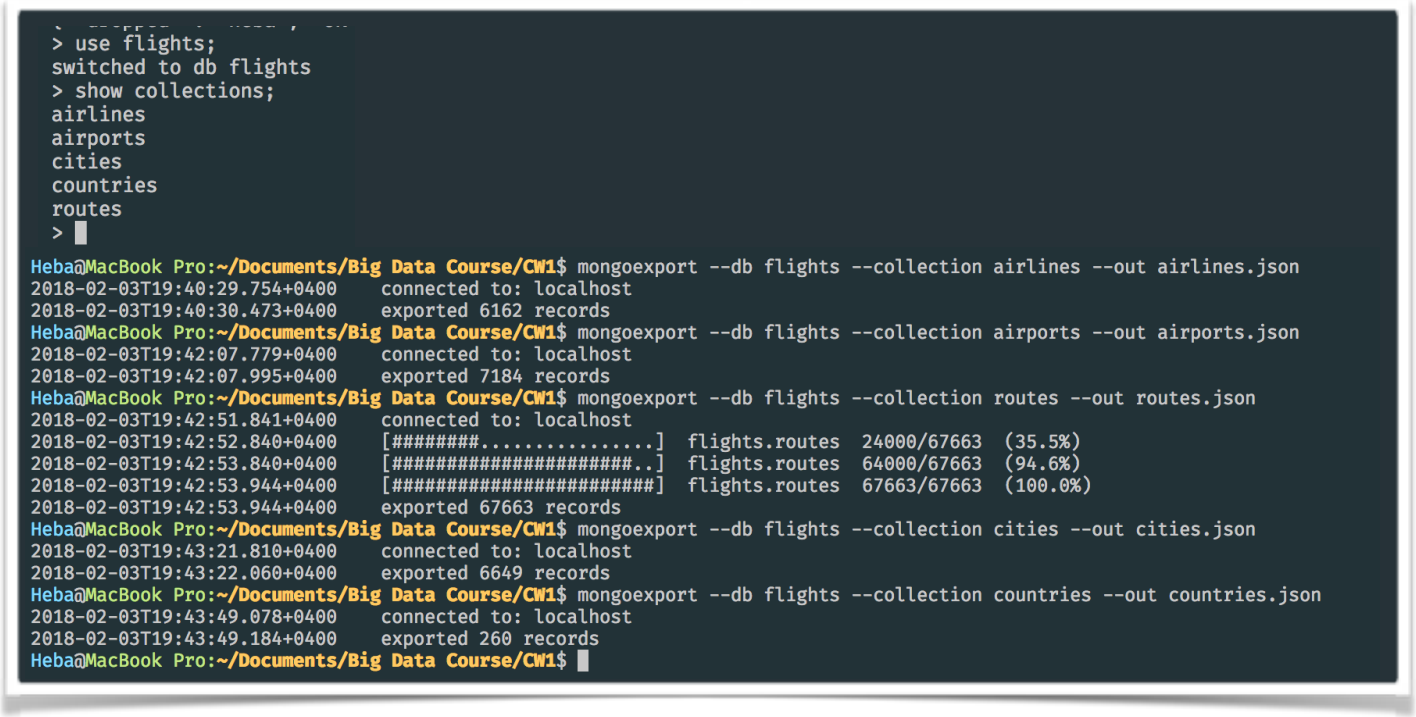
Mongify is a command line tool to automate the ETL Process of migrating from SQL to *MongoDB*, built in Ruby, that takes in as arguments a database configuration file for connecting to both *MySQL* and *MongoDB,* and a translation file that defines different columns and how they should be mapped and copied into the new *MongoDB* database.

*All source code files are attached in the appendix.

**All source code and JSON files generated as *mongoexport* command output are uploaded to the following *Github Repository:* https://github.com/HebaNAS/Big-Data-1



Checking that our data has been copied successfully, I exported then to JSON files.



The **Transformation** process is explained in the *translation* file attached in the appendix.
Basically, I defined the columns present in the current *MySQL* schema along with their types and the relationships they have with other tables.
I also instructed the tool to embed some documents (after translation from tables) inside other collections on the field defined as reference to the parent collection.

A snapshot of the resulting files from the export process:

Task 5

Limitations of current approach with respect to its scalability

• Updating/appending new records into the collections will need to carry the same process for the documents that are embedded inside others. For instance, for the **routes** embedded inside the **Airlines** collection, any update to the parent **Routes** collection will need to be followed by updating those records that were embedded as subdocuments inside the **Airlines** collection.

I tried to use **ObjectID** whenever possible as embedded object to refer back to the document in its main collection, to allow for easily updating and appending records (as the **ObjectID** is just a pointer and doesn't need to be updated itself).

• Maximum document size is 16MB. This constraint is to make sure than no single document can take up too much RAM on the system. However, if datasets are in high volumes and there are many embedded arrays of objects inside the document, it may exceed this limit.

• Maximum nesting inside one document is 100 levels. This is not an easily hit constraint, having a hierarchy of 100 sub-documents inside each other.

• Maximum nodes in a replica set is 50 (was 12 before).

• Maximum connections number to one database is hardcoded to 20,000, it can be changed though (through configuration files or by passing as an argument in the command line while starting the *mongod* server).

• On linux, one *MongoDB* instance can't store more than 64TB of data (this is not a huge issue), but on Windows, this limit falls down to only 4TB of data for each instance/node.

Task 6

Dataset Analysis

Query 1: Get all Airlines in Country in a sorted fashion

```
db.countries.aggregate(
 {
        "$match": { "name": "United Arab Emirates"}
 }, {
        "$unwind": "$airlines"
 }, {
        "$sort": {"airlines.name": 1}
 }, {
        "$project": {"airlines.name": 1, "_id": 0
 })
```

- I queried against the **countries** collection.

- I used **aggregate** for extracting the results.

- I specified the field I need to match (name of the country, UAE in this query).

- The results will be extracted from **airlines** embedded document (array of sub-documents), I used **unwind** to flatten the array and create a new document for each item in the airlines array. This step was needed before sorting.

- I sorted the results in an ascending order using the airport name field.

- I wanted to extract only the airport names, so I use **project** to remove all other fields from the results and return just the airport names field, I used zero for id field value because ids are returned by default and I did not want them in the result.

Results of that query (I used the —*eval* flag with mongo command in the terminal to output results to JSON file):


```
[
      {
            "airlines" : {
                  "name" : "Abu Dhabi Amiri Flight"
            }
      },
      {
            "airlines" : {
                  "name" : "Aerovista Airlines"
            }
      },
      {
            "airlines" : {
                  "name" : "Aerovista Gulf Express"
            }
```

```
	},
	{
		"airlines" : {
			"name" : "Air Arabia"
		}
	},
	{
		"airlines" : {
			"name" : "Al Rais Cargo"
		}
	},
	{
		"airlines" : {
			"name" : "Ave.com"
		}
	},
	{
		"airlines" : {
			"name" : "Avjet International (FZE)"
		}
	},
	{
		"airlines" : {
			"name" : "Cargo Plus Aviation"
		}
	},
	{
		"airlines" : {
			"name" : "Dubai Airwing"
		}
	},
	{
		"airlines" : {
			"name" : "Eastern Sky Jets"
		}
	},
	{
		"airlines" : {
			"name" : "Elite Jets"
		}
	},
	{
		"airlines" : {
			"name" : "Emirates"
		}
	},
	{
		"airlines" : {
			"name" : "Etihad Airways"
		}
	}
]
```


This is only part of the results, for full results please refer to https://github.com/HebaNAS/Big-Data-1/blob/master/airlines_in_uae.json.

Query 2: Get the name of a serving City of a specific Airport

```
db.cities.find({"airports.iata": {$eq: "DXB"}).pretty()
```

- I queried against the **cities** collection.

- I specified the field I need to filter by (iata code of the airport, DXB in this query).

Results of that query (I used the —*eval* flag with mongo command in the terminal to output results to JSON file):


```
[
	{
		"_id" : ObjectId("5a75b8cea9f575338d01439a"),
		"name" : "Dubai",
		"country" : ObjectId("5a75b8cea9f575338d013c74"),
		"timezone" : 4,
		"tz_id" : "Asia/Dubai",
		"airports" : [
			{
				"name" : "Dubai International Airport",
				"iata" : "DXB",
				"apid" : ObjectId("5a75b8c4a9f575338d002047")
			},
			{
				"name" : "Al Maktoum International Airport",
				"iata" : "DWC",
				"apid" : ObjectId("5a75b8c4a9f575338d002f03")
			}
		]
	}
]
```

Query 3: Get a list of airport in a specific geographic area

```
db.airports.find({"y": {$gt: 22, $lt: 33}, "x": {$gt: 35, $lt: 60}})
```

- I queried against the **airports** collection.

- I specified the field we need to filter by: **x** and **y** (longitude and latitude) of airports, I specified a range of values to search within. Here I searched for airports in the Middle East. I limited the values to search for to any latitude (**y**) greater than (**$gt**) 22 and less than (**$lt**) 33, I did the same for longitude (**x**).

Results of that query (I used the —*eval* flag with mongo command in the terminal to output results to JSON file). Only part of the results are shown here, for the full results please refer to the following file https://github.com/HebaNAS/Big-Data-1/blob/master/list_of_airports_in_lat_lng.json :


```
[
	{
		"_id" : ObjectId("5a75b8c4a9f575338d001fe3"),
		"name" : "Jubail Airport",
		"city" : ObjectId("5a75b8cea9f575338d0147c0"),
		"icao" : "OEJB",
		"x" : 49.40510177612305,
		"y" : 27.038999557495117,
		"elevation" : 26
	},
	{
		"_id" : ObjectId("5a75b8c4a9f575338d001fe6"),
		"name" : "King Khaled Military City Airport",
		"city" : ObjectId("5a75b8cea9f575338d0148ad"),
		"icao" : "OEKK",
		"x" : 45.528198,
		"y" : 27.9009,
		"elevation" : 1352
	}
]
```

Query 4: Get a list of destinations that a specific airline flies to

```
db.airlines.distinct({"routes.dst_ap": {"alias": "Emirates Airlines"}})
```

- I queried against the **airlines** collection.

- I specified the field we need to filter by: airline **alias** or full name (Emirates Airlines in this query)

- To get a set of results, i.e., have no duplicates, I used (**distinct**) command in *mongo*. It filters that results further using the destinations of airlines extracted from the embedded routes sub-document (**routes.dst_ap**).

Results of that query (I used the —*eval* flag with mongo command in the terminal to output results to JSON file):


```
[ "ABJ", "ACC", "ADD", "ADL", "AKL", "ALG", "AMD", "AMM", "AMS", "ARN", "ASP",
"ATH", "BAH", "BCN", "BEY", "BGW", "BHX", "BKK", "BKQ", "BLR", "BNE", "BOM", "BOS",
"BSR", "CAI", "CAN", "CCJ", "CCU", "CDG", "CGK", "CHC", "CMB", "CMN", "COK", "CPH",
"CPT", "DAC", "DAR", "DEL", "DFW", "DKR", "DME", "DMM", "DOH", "DUB", "DUR", "DUS",
"DXB", "EBB", "EBL", "EZE", "FCO", "FRA", "GIG", "GLA", "GOV", "GRU", "GVA", "HAM",
"HKG", "HKT", "HND", "HRE", "HYD", "IAD", "IAH", "ICN", "IKA", "ISB", "IST", "JED",
"JFK", "JNB", "KBL", "KBP", "KHI", "KIX", "KRT", "KUL", "KWI", "LAD", "LAX", "LCA",
"LED", "LGW", "LHE", "LHR", "LIS", "LOS", "LRE", "LUN", "LYS", "MAA", "MAD", "MAN",
"MCT", "MED", "MEL", "MLA", "MLE", "MNL", "MRU", "MUC", "MXP", "NBO", "NCE", "NCL",
"NRT", "PEK", "PER", "PEW", "PRG", "PVG", "ROK", "RUH", "SAH", "SEA", "SEZ", "SFO",
"SGN", "SIN", "SKT", "SYD", "TIP", "TPE", "TRV", "TUN", "VCE", "VIE", "WAW", "WLG",
"YYZ", "ZQN", "ZRH" ]
```

Query 5: Get a list of destination airports that start their journey from a specific airport

This query had to span 2 collections, I had to write 2 queries for both collections, so I had to write javascript code to store the results from each query. This is one of *MongoDB*'s limitations, it's unable to perform joins or query across multiple collections.

I used *mongo*'s command line to run the code inside the file and write the output to another file as JSON.

```
mongo flights command.js > list_of_dst_src_dxb.json
```

*Full Javascript Code can be found here https://github.com/HebaNAS/Big-Data-1/blob/master/command.js , code can also be found in the Appendix of this report.

Results of that query:


```
[
	[{ "name" : "Amsterdam Airport Schiphol" }],
	[{ "name" : "Charles de Gaulle International Airport" }],
	[{ "name" : "General Edward Lawrence Logan International Airport" }],
	[{ "name" : "Dallas Fort Worth International Airport" }],
	[{ "name" : "Washington Dulles International Airport" }],
	[{ "name" : "George Bush Intercontinental Houston Airport" }],
	[{ "name" : "John F Kennedy International Airport" }],
	[{ "name" : "Los Angeles International Airport" }],
	[{ "name" : "Malpensa International Airport" }],
	[{ "name" : "Seattle Tacoma International Airport" }],
	[{ "name" : "San Francisco International Airport" }],
	[{ "name" : "London Heathrow Airport" }]
]
```


This is only a part of the results, for full results please refer to https://github.com/HebaNAS/Big-Data-1/blob/master/list_of_dst_src_dxb.json

Query 6: Update one record

```
db.routes.updateOne({dst_ap: "KZN"}, {$set: {"stops": 1}})
```

- I updated only one record in **routes** collection.

- I extracted one record of the set that matches the criteria of a destination airport code of "KZN".

- I set a new value for the field "stops" to 1.

The result of that query is

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

When using find to see the result in the collection:

```
db.routes.find({})
```

This will give us the first few documents in the collection, we will find the first one's "stops" field has been changed to the value of 1.

Appendix

- *Mongify* command line tool installation:

```
gem install mongify
```

- Database configuration (database.config):

```
# Creating a connection to MySQL Database
sql_connection do
  adapter   "mysql"
  host      "127.0.0.1"
  username  "root"
  password  "toor"
  database  "flights_database"
end

# Creating a connection to MongoDB
mongodb_connection do
  host      "127.0.0.1"
  database  "new"
end
```

- *Mongify* translation code (database.translation.rb):

```
# Copying Airlines Table from MySQL into a MongoDB Collection
# Airlines Table will be embedded as a subdocument inside Countries Collection
# identified by the country field which is a foreign key that refernces that country
# in Countries Table
table "airlines", :embed_in => :countries, :on => :country do
  column "alid", :integer, :references => :airlines
  column "name", :text
  column "iata", :string
  column "country", :string, :references => :countries
end

# Copying Airlines Table from MySQL into a MongoDB Collection
table "airlines" do
  column "alid", :key, :as => :integer
  column "name", :text
  column "iata", :string
  column "icao", :string
  column "callsign", :text
  column "country", :string, :references => :countries
  column "alias", :text
  column "mode", :string
  column "active", :string
end

# Copying Airports Table from MySQL into a MongoDB Collection
# Airports Table will be embedded as a subdocument inside Cities Collection
# identified by the city field which is a foreign key that refernces that city in
# Cities Table
table "airports", :embed_in => :cities, :on => :city do
  column "apid", :integer, :references => :airports
  column "name", :text
  column "city", :integer, :references => :cities
  column "iata", :text
end
```

```
# Copying Airports Table from MySQL into a MongoDB Collection
table "airports" do
  column "apid", :key, :as => :integer
  column "name", :text
  column "city", :integer, :references => :cities
  column "iata", :key, :as => :string
  column "icao", :string
  column "x", :float
  column "y", :float
  column "elevation", :integer
end

# Copying Routes Table from MySQL into a MongoDB Collection
# Routes Table will be embedded as a subdocument inside Airlines Collection
# identified by the airline id (alid) field which is a foreign key that refernces
# that airline in Airlines Table
table "routes", :embed_in => :airlines, :on => :alid do
  column "alid", :integer, :references => :airlines
  column "rid", :integer, :references => :routes
  column "src_ap", :string
  column "dst_ap", :string
  column "stops", :text
end

# Copying Routes Table from MySQL into a MongoDB Collection
table "routes" do
  column "rid", :key, :as => :integer
  column "alid", :integer, :references => :airlines
  column "src_ap", :string, :references => :airports
  column "src_apid", :integer, :references => :airports
  column "dst_ap", :string, :references => :airports
  column "dst_apid", :integer, :references => :airports
  column "codeshare", :text
  column "stops", :text
  column "equipment", :text
end

# Copying Countries Table from MySQL into a MongoDB Collection
table "countries" do
  column "code", :key, :as => :string
  column "name", :text
  column "city", :integer, :references => :cities
  column "oa_code", :string
  column "dst", :string
end

# Copying Cities Table from MySQL into a MongoDB Collection
# Cities Table will be embedded as a subdocument inside Countries Collection
# identified by the country field which is a foreign key that refernces that country
# in Countries Table
table "cities", :embed_in => :countries, :on => :country do
  column "cid", :integer, :references => :cities
  column "name", :text
  column "country", :string, :references => :countries
  column "tz_id", :text
  column "airports", :integer, :references => :airports
end

# Copying Cities Table from MySQL into a MongoDB Collection
table "cities" do
  column "cid", :key, :as => :integer
  column "name", :text
  column "country", :string, :references => :countries
```

```
  column "timezone", :float
  column "tz_id", :text
end
```

- Javascript code for Query 5 (list_of_dst_src_dxb.js):

```
// Find _ids for matching routes where the src airport is DXB
var matches = db.routes.find({src_ap: "DXB"}, {"dst_apid": 1, _id: 0});

// Store matching _id values into an array of ObjectIDs
var dst = [];
matches.forEach(function(match) {
  if (match.dst_apid != null) {
    dst.push(match.dst_apid);
  }
});

// Iterate through the array of ObjectIDs holding matching values of the previous
// query and use each id to find the matching airport, then get the name of that
// airport
var results = [];
dst.forEach(function(one) {
  results.push(db.airports.find({"_id": one}, {"name": 1, "_id": 0}).toArray());
});

// Print results as json
printjson(results);
```

All code can be found here: https://github.com/HebaNAS/Big-Data-1/

References

1. A.J.G. Gray, H. Zantout. "CAP Theorem and DBMS Landscape," Lecture Notes. Heriot-Watt University.
2. A.J.G. Gray, H. Zantout. "Data Stores," Lecture Notes. Heriot-Watt University.
3. T. Wellhausen, "Highly Scalable , Ultra-Fast and Lots of Choices A Pattern Approach to NoSQL," in Companion Proceedings of the 17th European Conference on Pattern Languages of Programs, 2012, pp. B1–1–B1–9.
4. Lourenço, J. R., Cabral, B., Carreiro, P., Vieira, M., & Bernardino, J. (2015). "Choosing the right NoSQL database for the job: a quality attribute evaluation." Journal of Big Data, 2(1), 18.
5. Brewer, E. (2012). "CAP twelve years later: How the "rules" have changed." Computer, 45(2), 23–29.
6. Dale Kim. "NoSQL technologies are built to solve business problems, not just "wrangle big data"."
7. MongoDB Docs - Data Model Design. "https://docs.mongodb.com/manual/core/data-model-design/."
8. MongoDB Docs - Model Relationships Between Documents. "https://docs.mongodb.com/manual/applications/data-models-relationships/ ."
9. MongoDB Docs - MongoDB Limits and Thresholds. "https://docs.mongodb.com/manual/reference/limits/#mongodb-limits-and-thresholds ."
10. MongoDB Docs - Update Documents. "https://docs.mongodb.com/manual/tutorial/update-documents/#update-documents."
11. MongoDB Docs - Aggregation. "https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/#aggregation-pipeline-stages."
12. Modeling Data for NoSQL Document Databases. "https://youtu.be/-o_VGpJP-Q0."
13. Matt Allen. "Relational Databases Are Not Designed For Scale," Blog Article.
14. Philipp Hauer. "Why Relational Databases are not the Cure-All. Strength and Weaknesses," Blog Article.
15. Mongify. "http://mongify.com/."
16. Mongify Docs. "http://www.rubydoc.info/gems/mongify/."