NoSQL Databases

Lacey Conrad
MSDS 610
Regis University

May, 2020

1 Introduction

As we have learned over the last couple of weeks, structured data is easily managed in a relational database using a predefined schema, which dictates the manner in which the data are stored and retrieved. However, the relational model struggles with semi- and unstructured data, and in particular, Big Data. Enter NoSQL databases, which are capable of storing structured and semi-structured data without the need of writing a schema (Pal, 2016).

NoSQL has garnered a great deal of popularity and use lately because of the vast amount of unstructured data that is now being housed in databases. Vast quantities of Big Data are produced daily, and this data is primarily unstructured. This frequent production of data simply didn't exist several decades ago when relational models were all we needed. The NoSQL database model evolved to manage this unstructured data in a non-relational, more flexible way, which included no tabular relations or schema (Pal, 2016).

The assignment for this week included the installation and basic usage of a NoSQL database, MongoDB. First, I downloaded and installed MongoDB and Mongo Compass locally on my PC following the instructions provided in the lab handouts. Then, I downloaded the Los Angeles Parking Citations data set from Kaggle. I used the MongoDB Compass GUI to create the traffic citations database, and then imported the Parking Citations data set. After practicing creating queries using the Compass query bar, I moved on to the next portion of the lab, connecting Python with MongoDB using Jupyter Notebook. Since I already had Anaconda installed (which includes Jupyter), I was able to skip some of the instructions and attempted to establish a connection between Python and Jupyter. Once I was able to do this, I began to write queries and practice using Pandas to plot my data.

2 Written portion - Questions from week 4

1. Why is NoSQL useful? What are use cases for NoSQL?

   NoSQL is an alternative to relational databases. It is non-relational, model-less, and table-less. This allows for easier data management, at the cost of standardization and ACID compliance. Several common uses of NoSQL are listed below:

   (a) Dealing with Big Data: Typical relational databases are only capable of vertical scaling, which is limiting and costly. Most Big Data scales best horizontally, which is either not easy or impossible to do in a relational database. NoSQL databases were designed with Big Data in mind.

   (b) Storing semi- and unstructured data: A lot of recent data is semi-structured, and developing relational database schemas for it is very difficult. NoSQL databases have a multitude of ways to deal with data that isn't as structured as relational databases require.

   (c) Supporting high-availability services: Many of the popular NoSQL database engines were developed with the idea of high-availability services in mind. They support clustering and are able to prevent the loss of data even when several nodes in a cluster fail.

(d) Implementing caching layers: NoSQL key-value type databases excel at caching applications (Robles, 2018).

I also found some enterprise use cases, which are particularly interesting:

(a) Personalization: In order to make a users experience online more personalized, a great deal of data is required. NoSQL databases scale elastically and can build and update user profiles as they change in real-time.

(b) Real-Time Big Data: Being able to work with data as it is generated allows companies to operate more efficiently, enabling decisions to be make immediately on current data. NoSQL can be used as a front-end and back-end data management engine.

(c) Mobile Applications: Mobile applications face a great deal of scalability challenges as they often start off with a small number of users and expand as the product receives publicity. This manner of starting with a small deployment and expanding as you go is the type of scalability NoSQL databases were developed for (Stephan, 2015).

2. What are some of the different NoSQL databases, and in what cases are they useful?

(a) MongoDB (Document-oriented): Free and open-source. Uses sharding to scale databases horizontally. MongoDB is very popular in JavaScript frameworks. Data is stored in JSON-like documents, and any field can be indexed in a document. Supports dynamic schemas ("Best NoSQL databases 2020 - Most popular among programmers," 2020).

(b) Cassandra (Wide column/column family): This database was developed at Facebook for an inbox search. It is able to handle very large amounts of structured data through a distributed, decentralized, column-oriented data storage system. Supports ACID properties. Also supports MapReduce with Hadoop ("Best NoSQL databases 2020 - Most popular among programmers," 2020). Optimized for clusters, in particular, clusters across multiple datacenters (Mayo, 2016).

(c) HBase (Wide column/column family): Designed by Google for the BigTable database. Free and open-source. It is a distributed database. Capacity can be added by adding more servers. HBase integrates with Hadoop as a source and a destination ("Best NoSQL databases 2020 - Most popular among programmers," 2020). Very efficient database for sparse, distributed data. Current implementations of HBase include LinkedIn, Facebook, and Spotify (Mayo, 2016).

(d) Neo4J (Graph): Uses pointers to navigate and traverse the graph. Supports full ACID rules. Easy query language. Manages semi-structured data very well ("Best NoSQL databases 2020 - Most popular among programmers," 2020). Has an open-source implementation. Advantageous in data mining and pattern recognition scenarios (Mayo, 2016).

(e) redis (Key-value pairs): Redis is an in-memory but persistent on-disk database. Foregoing data writes creates incredibly fast performance. It supports a huge variety of data types and has support for many programming languages. All operations are atomic. Redis also provides Redis Sentinel, which replicates the data into a distributed system ("Best NoSQL databases 2020 - Most popular among programmers," 2020).

3. What are pros and cons of NoSQL vs SQL?

NoSQL

(a) Pros:

i. Scalable/Flexible: The NoSQL model allows for easy scaling horizontally, which is currently favored (and is more advantageous for Big Data) over vertical scalability (Miller, 2017).

ii. Able to store massive amounts of data: Not only are NoSQL databases able to store unlimited amounts of data, these data can also be without any structure (Brody, 2017).

iii. Can be used with cloud computing: Cloud-based storage requires data to be spread across multiple servers, and NoSQL databases were designed for managing data like this (Brody, 2017).

iv. Less prep work: NoSQL databases utilize schema-on-read, and as such, does not slow down development in the manner relational databases can since less time is spent designing the database (Brody, 2017).

v. Low cost: NoSQL databases do not rely on the costly infrastructure relational databases do, and since NoSQL is tolerant of having a certain amount of node failure, databases can be housed on affordable hardware (Brody, 2017).

(b) Cons:

i. Less support: While SQL vendors have established support that is available 24 hours a day, this is not yet guaranteed by NoSQL vendors (Miller, 2017).

ii. Less analytic features: NoSQL was created with web 2.0 applications in mind. NoSQL has focused on meeting the demands for these applications, at the cost of offering less analytic features (Miller, 2017).

SQL

(a) Pros

i. ACID compliancy: The ACID characteristics (atomicty, consistancy, isolation, and durability) protects the integrity of the database and reduces anomalies.

ii. Data is structured and inflexible: Some companies are not growing their databases at a rapid pace, and have little use for a database that can work with different data types of high data volume (Brody, 2017).

iii. Age: SQL has been around for a very long time. It is a proven technology with plenty of developer experience and an abundance of support (Brody, 2017).

iv. Joins: SQL has the ability to join two or more tables together through Joins. NoSQL has no simple analog to this (Brody, 2017).

(b) Cons:

i. Interface: Interfacing can be complex (K, 2019).

ii. Expense: The expenses involved in SQL operations are very high, both for the infrastructure and for the expertise to manage the infrastructure (K, 2019).

iii. Implementation: It is not uncommon to encounter proprietary extensions in certain databases to ensure a vendor lock-in(K, 2019).

iv. Control: There are hidden rules and conditions in SQL databases, and programmers do not have full power over the database for this reason(K, 2019).

4. What did you find from the MongoDB queries? (e.g. what were the most expensive tickets)

   I found the most expensive parking tickets (all 10 returned by my query) to be for `$98`, and had a Violation Description of PK OVERSIZE. A little research seems to indicate that this violation is for parking an over-sized vehicle without a permit.

   When excluding California, the license plate query showed vehicles from Nevada (NV), Texas (TX) and Arizona (AZ) receiving the most parking citations. Not surprisingly, the percentage of total citations received by vehicles from each state changed depending on how many results you limited the query by. I started with a limit of 100, and then tried 10,000 and realized I got significantly different percentages for each state. Therefore, I reran the query with all of the data. I found that Arizona received 5.75% of the non-California parking violations, followed by Texas (9.23%), Nevada (8.64%), Florida (6.37%), and Washington (5.75%). When California was included, the percentage for each non-CA state was below 1%.

3 Methods and Code

3.1 MongoDB installation and data importation

First, I went to `Kaggle.com` and downloaded the "Los Angeles Parking Citations" data set used in this weeks exercise. I made sure to download the CSV file, and saved it in an easy to find location.

I then followed the week 4 lab supplemental to install MongoDB and MongoDB Compass locally to my PC. Once the installation was complete, I started up the Compass GUI. I created the la_traffic database and la_traffic_citations collection in Compass. I clicked "create," and then clicked on the name of the collection under the la_traffic database header. I selected collection, and then import data. After highlighting the CSV option, I located the CSV file I had downloaded from `Kaggle.com` and clicked on the import button.

After the data set was imported into MongoDB via Compass, I used the Compass gui to learn some basic MongoDB queries. After some practice, I was able to perform the first query required for this weeks assignment via the Compass GUI with the following added to the blank query fields:

Options	**Command**
Project	`{"Ticket number":1, "Violation Description":1, "Fine amount":1}`
Sort	`{"Fine amount":-1}`
Maxtimems	30000
Limit	10

Table 1: Writing the query for the 10 most expensive tickets in the Compass GUI.

3.2 Connecting to MongoDB in Python and writing queries

Our lab documents recommended using Python via Jupyter installed alongside Anaconda. Fortunately, I already had Anaconda installed and had basic familiarity with Jupyter, therefore to set up MongoDB on Jupyter all I had to do was verify `pymongo` was installed. I did this with the Anaconda prompt. With MongoDB running, I proceeded to run Jupyter notebooks.

I made sure I was connected correctly to MongoDB before diving deeply into writing queries for the assignment. I added the following lines of code to connect Python to my MongoDB database, and to make sure I can access the first document from the database. All code from this point forward will be commented to show/explain the process.

```
from pymongo import MongoClient

# Allowing Python access to the MongoDB database:
client = MongoClient()

#Connecting the client to the MongoDB database "la_traffic":
db=client['la_traffic']

# Creates the MongoDB collection "tickets" from the MongoDB collection
# "la_traffic_citations":
tickets=db['la_traffic_citations']

# Returns the first document in the tickets collection:
tickets.find_one()
```

I received the following output from `tickets.find_one()`

Table 2: The output of running the Python code `tickets.find_one()` on the parking citation database.

tickets.find_one()
'_id': ObjectId('5ecbea08a1360440c4dcd9d3'),
'Ticket number': '1103341116',
'Issue Date': '2015-12-21T00:00:00.000',
'Issue time': '1251',
'Meter Id': '',
'Marked Time': '',
'RP State Plate': 'CA',
'Plate Expiry Date': '200304',
'VIN': '',
'Make': 'HOND',
'Body Style': 'PA',
'Color': 'GY',
'Location': '13147 WELBY WAY',
'Route': '01521',
'Agency': '1',
'Violation code': '4000A1',
'Violation Description': 'NO EVIDENCE OF REG',
'Fine amount': '50',
'Latitude': '99999',
'Longitude': '99999',
'Agency Description': '',
'Color Description': '', 'Body Style Description': ''

Taking this as verification that I had correctly connected Python (via Jupyter) to the parking citation data set I had created in MongoDB, I decided it was safe to begin work on the queries required for this weeks lab. I found a few places that mapped queries SQL to MongoDB queries, and decided to pull from one of these charts several of the queries I thought would be helpful. A few important queries I found:

Table 3: SQL to MongoDB conversion, adapted from `docs.mongodb.com/manual/reference/sql-comparison/`.

SQL Code	**MongoDB Code**
SELECT * FROM people	db.people.find()
SELECT id, user_id, status FROM people	db.people.find(`{},{user_id:1,status:1}`)
SELECT * FROM people WHERE status != "A"	db.people.find(`{status:{$ne: "A"}}`)
SELECT * FROM people WHERE status = "A" ORDER BY user_id DESC	db.people.find(`{status:"A"}).sort({user_id:-1})` (*for sort(), ASC = 1, DESC = -1)

These MongoDB queries, with a few changes, helped me write the queries for this assignment.

Continuing on (with `docs.mongodb.com/manual/reference/sql-comparison/` open for reference), I started on the first query: what are the amounts and violation descriptions for the top 10 most expensive tickets. The following is the query I wrote for this question:

```
1 # Using the pymongo.cursor.Cursor datatype to store the 100 most expensive tickets,
```

```
# in addition to the id, ticket number, violation description, and fine amount
# of each record from the most expensive ticket to the least
# (In sort(), -1 is a descending sort, whereas 1 is an ascending sort).
fine_sort = tickets.find({},{"Ticket number":1, "Violation Description":1,
        "Fine amount":1}).sort("Fine amount", -1).limit(100)
```

As one of the goals of this lab exercise was to view the results of the queries using pandas and matplotlib, I went ahead and imported those tools, and then created a dataframe out of the data stored in `fine_sort`. A chart showing the results of this query will be located in the results section.

```
# Importing Pandas and matplotlib
import pandas as pd
import matplotlib.pyplot as plt
# Auto-show plots in the notebook:
%matplotlib inline

# Most expensive ticket query (explained above):
fine_sort = tickets.find({},{"Ticket number":1, "Violation Description":1,
        "Fine amount":1}).sort("Fine amount", -1).limit(100)

# Creates a Pandas dataframe out of the fine_sort data:
df3 = pd.io.json.json_normalize(fine_sort)

# Displays the first 10 results in the df3 dataframe.
# In this case, this returns the 10 most expensive tickets:
df3.head(10)
```

Next, I moved onto the query to locate all the records of citations based on the state of the vehicles license plate, excluding California. Additionally, I used Pandas to plot a bar chart to plot the frequency with which each state received citations. The code is as follows:

```
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# A query to return citations from vehicles without California
# license plates.  I initially limited the query to 100 results, to
# speed up the query.  Later on, I ran the query producing the
# entirity of the results, since the results with .limit(100)
# were unlike the overall results.
no_ca = tickets.find({'RP State Plate':{'$ne': 'CA'}}).limit(100)

# Creates a Pandas dataframe out of the no_ca data:
df2 = pd.io.json.json_normalize(no_ca)

# The following lines of Python create a barchart including
# a title and labeled axes out of the Pandas datafram df2.
no_ca_chart = df2['RP State Plate'].value_counts()[:20].plot.bar()
plt.tight_layout()
no_ca_chart.set_xlabel("State")
no_ca_chart.set_ylabel("Number of Violations")
no_ca_chart.set_title("Parking citations given in California by state
        (excluding California)")

# Saving and downloading the plot to a local directory:
fig = no_ca_chart.get_figure()
fig.savefig('no_ca.jpg')
```

4 Results and Output

The following table are the results of the query written to find the 10 most expensive tickets. The fine amount for each of these violations is $98, and the violation description is listed as "PK OVERSIZ," which may mean a violation for parking an over-sized vehicle without a permit.

	_id	Ticket number	Violation Description	Fine amount
0	5ecbeda3a1360440c4710b31	4357748430	PK OVERSIZ	98.0
1	5ecbeda5a1360440c47162c5	4357851842	PK OVERSIZ	98.0
2	5ecbeda6a1360440c4717e36	4359460033	PK OVERSIZ	98.0
3	5ecbeda6a1360440c471821f	4360048254	PK OVERSIZ	98.0
4	5ecbeda6a1360440c4718224	4360048265	PK OVERSIZ	98.0
5	5ecbeda9a1360440c472075b	4359446453	PK OVERSIZ	98.0
6	5ecbedaaa1360440c47212c2	4359346530	PK OVERSIZ	98.0
7	5ecbedb3a1360440c4736e12	4504035281	PK OVERSIZ	98.0
8	5ecbedb3a1360440c47371d2	4504283582	PK OVERSIZ	98.0
9	5ecbea0da1360440c4dd8723	4274078911	PK OVERSIZ	98

Table 4: Basic information from the 10 most expensive tickets in the LA parking citation data set.

As is visible in the following figure, the license plate query showed vehicles from Arizona (AZ), Texas (TX), and Nevada (NV) receiving the most parking citations when California was excluded.

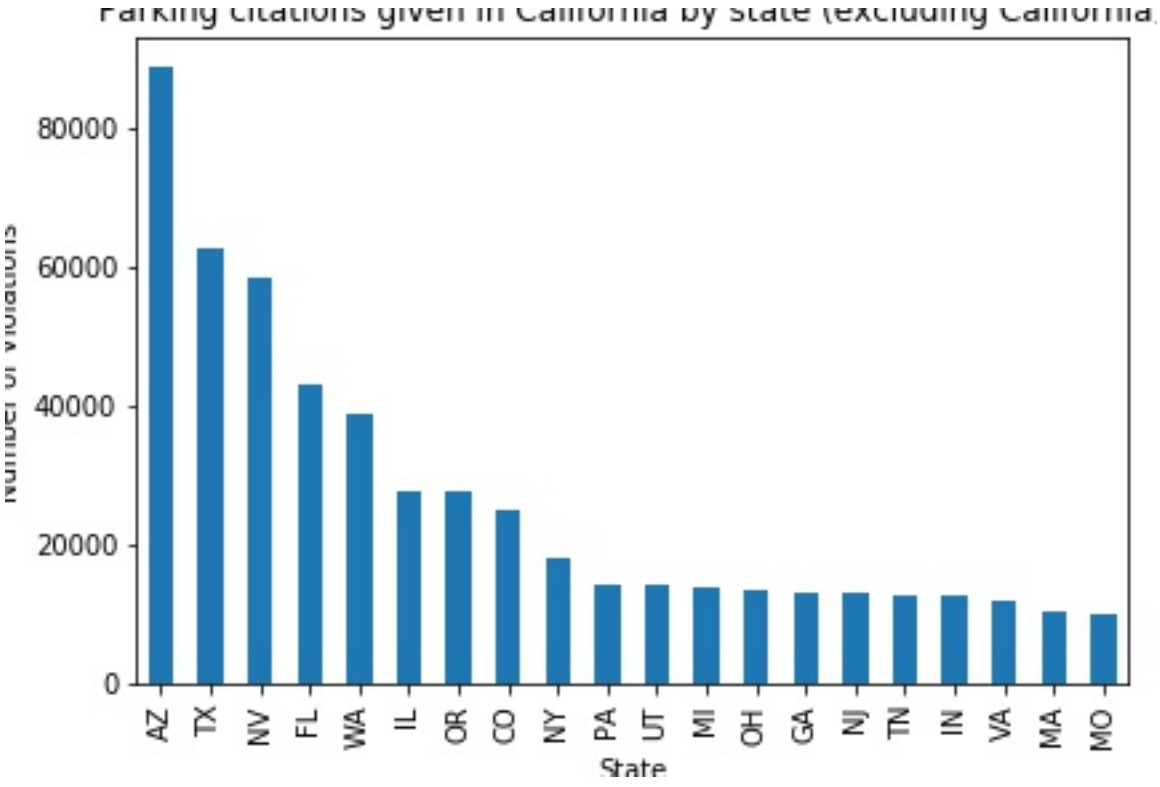

Figure 1: Tickets from parking violations in California organized by state of license plate, excluding tickets issued to cars with California license plates. (n=678,214)

The percentage of total citations from each state changed depending on how many results you limited the query by. I found that when I limited my search to 100 results I received a different state ranking, and different percentage of citations per state than when I limited by 10,000. Therefore, I reran the query to show all of the data. I found that Arizona received 5.75% of the non-California parking violations, followed by

Texas (9.23%), Nevada (8.64%), Florida (6.37%), and Washington (5.75%). When California was included, the percentage for each non-CA state was below 1%.

Rank	State	Number	% of total violations	% of non-CA violations
-	*California (CA)*	*9,203,628*	*93%*	-
1	Arizona (AZ)	88657	0.89%	13.1%
2	Texas (TX)	62654	0.63%	9.23%
3	Nevada (NV)	58672	0.59%	8.64%
4	Florida (FL)	43191	0.44%	6.37%
5	Washington(WA)	39060	0.39%	5.75%

Table 5: Percentage of violations for the top five most frequently encountered state license plates, excluding California.

5 Analysis

5.1 Data analysis

The data from the top 10 most expensive tickets shows that they were all capped at the same price, for the same violation. I went back to the Kaggle, and then to the location where the data sets are generated from the city of Los Angeles, and I could not find much in the way of explanations for what some of the less obvious fields mean. However, in the case of the most expensive tickets, they were all issued for a violation of "PK OVERSIZ."

Parking tickets issued by state of license plate was a very interesting data set to look at, especially when California was removed. As expected, cars registered in California made up over 90% of the parking tickets issued. When California was removed, I was able to see what states made up the most out of state parking tickets issued. Arizona, one of the states bordering California (and in close proximity to LA), received the most tickets, topping Texas by 20,000 tickets. I was surprised to see Texas and Florida in the top 5 list, as both states are a fairly long distance away from California. Also, Texas came in second, ahead of another of California's neighbors, Nevada.

5.2 Analysis of NoSQL

My understanding of MongoDB is that the data we are working with in this database are stored in a series of documents. The structure of this database, with each document being a parking citation, and those documents being stored in a collection, really illustrates a document-oriented NoSQL database.

Additionally, I can imagine the benefits to allowing fields to be left empty to be significant. In SQL, I know there is still "data" in empty columns since "null" must be included when no value is available. This still takes up valuable storage space. If you have a sparse data set in SQL, you could waste a great deal of storage space on "null" values. It can be seen in the following two examples from the LA citations data set that many of the fields of the document are blank. In practice, police officers likely do not need all the information on a violation in order to process the violation, and it would be impractical for them to waste time filling in fields which are unneeded or they are otherwise unable to fill in. Basically, police officers have a lot of other work to do instead of filling out a SQL database.

Table 6: Example of a document from the LA driving citations data set. Empty fields are in bold.

'_id': ObjectId('5ecbea09a1360440c4dce0bb'),
'Ticket number': '4273023300',
'Issue Date': '2015-12-30T00:00:00.000',
'Issue time': '904',
'Meter Id': 'SL413',
'Marked Time': '',
'RP State Plate': 'MD',
'Plate Expiry Date': '201607',
'VIN': '',
'Make': 'SUBA',
'Body Style': 'PA',
'Color': 'BL',
'Location': '3020 SUNSET BL',
'Route': '00444',
'Agency': '54',
'Violation code': '88.13B+',
'Violation Description': 'METER EXP.',
'Fine amount': '63',
'Latitude': '6478699.4',
'Longitude': '1852962.5',
'Agency Description': '',
'Color Description': '',
'Body Style Description': ''

Table 7: Another example of a document from the LA driving citations data set. Empty fields are in bold.

'_id': ObjectId('5ecbea08a1360440c4dcd9e7'),
'Ticket number': '1107780822',
'Issue Date': '2015-12-22T00:00:00.000',
'Issue time': '1105',
'Meter Id': '',
'Marked Time': '',
'RP State Plate': 'FL',
'Plate Expiry Date': '201611',
'VIN': '',
'Make': 'FORD',
'Body Style': 'PA',
'Color': 'WH',
'Location': '',
'Route': '2A1',
'Agency': '1',
'Violation code': '8069B',
'Violation Description': 'NO PARKING',
'Fine amount': '73',
'Latitude': '99999',
'Longitude': '99999',
'Agency Description': '',
'Color Description': '',
'Body Style Description': ''

6 Conclusion

When I first started using the MongoDB Compass GUI to query data in the parking citation database, I found it vague and lacking a sort of power that SQL queries seemed to possess. However, when I started using MongoDB with Python and Pandas I really started seeing the potential, and began enjoying the querying much more. I'm not sure why I enjoyed the queries more in Python, as they were more difficult to write, but they felt a bit closer to SQL. I felt like I once again had that power to slice away to the data I specifically wanted to look at.

That being said, the queries still felt a little clunky to write compared to SQL. In SQL, queries can be written with just a few words and a semicolon. Whether using the Compass GUI or writing queries in Python, MongoDB's style of writing queries proved to be more tedious, especially around the use of punctuation. And what worked in the Compass GUI did not necessarily work in Pymongo. In particular, I struggled writing the "most expensive ticket" query, until I did some research and realized I had to change out a colon for a comma in my `.sort("Fine Amount", -1)` statement.

Interestingly, I also saw the echo of many search engines found across the internet in the Compass GUI. Many times I have navigated to a website, for example a furniture website, and found myself entering various bits of information into several fields to search for my "ideal" couch. I am curious if those search pages are just user interface on top of a NoSQL database. While the code is clearly more difficult to write than a user entering a few bits of information into a search database, that would be the whole point of making a user-friendly interface.

7 References

1. *Best NoSQL databases 2020 - Most popular among programmers.* (2020, April 19). I'm Programmer. `https://www.improgrammer.net/most-popular-nosql-database/`

2. Brody, A. (2017, March 9). *SQL vs NoSQL: The differences explained.* Panoply Blog. https://blog.panoply.io/sql-or-nosql-that-is-the-question

3. K, J. (2019, August 1). *SQL and NoSQL advantages and disadvantages.* Web Solutions Blog. https://acodez.in/sql-and-nosql-an-overview/

4. Mayo, M. (2016, June). *Top NoSQL database engines.* KDnuggets. https://www.kdnuggets.com/2016/06/top-nosql-database-engines.html

5. Miller, B. (2017, September 27). *7 pros and cons of NoSQL.* Green Garage. https://greengarageblog.org/7-pros-and-cons-of-nosql

6. Pal, K. (2016, June 7). *Why the world is moving toward NoSQL databases.* Techopedia.com. https://www.techopedia.com/2/32000/trends/big-data/why-the-world-is-moving-toward-nosql-databases

7. Robles, P. (2018, August 22). *Five legitimate use cases for NoSQL databases.* Econsultancy. https://econsultancy.com/five-legitimate-use-cases-for-nosql-databases/

8. Stephan, T. (2015, October 30). *10 use cases where NoSQL will outperform SQL.* Network World. https://www.networkworld.com/article/2999856/10-use-cases-where-nosql-will-outperform-sql.html