ISOM 671: Managing Big Data – Assessment Part I

Sean Jung

There are 5 numbered questions and question number is likely to be correlated with the amount of effort needed to answer each question. Please refer to the notes at the end of this document. Please submit your responses as a single PDF or MS Word file by uploading it to course canvas page.

1. Answer the following short questions:

1.1. Briefly describe the requirements of a 3rd normal form (3NF).

(from pg. 483 of the textbook) According to Codd's original definition, a relation schema R is in 3rd normal form when it satisfies both two conditions 1) **if it satisfies 2NF** and 2) **If there's no transitive functional dependencies on the primary key.**

1.2. Briefly describe three differences between normalized modeling (OLTP database) and dimensional modeling (OLAP database).

https://www.guru99.com/oltp-vs-olap.html

OLTP	OLAP
OLTP supports transaction-oriented applications in a three-tier architecture (optimized for transactional superiority instead data analysis)	OLAP mainly is used for analyzing data stored in a database (E-commerce company can analyze purchases by its customers to tailor a personalized homepage with products which likely cater to their customer).
OLTP is useful to administer day to day transaction of an organization. The primary objective is data processing and not data analysis	OLAP creates a single platform for all type of business analysis that incorporates planning, budgeting, forecasting, analysis.
OLTP is characterized by large numbers of short online transactions (OLTP uses traditional DBMS)	OLAP is characterized by a large volume of data (data warehouse is created so that it can uniquely integrate different data sources for building a consolidated)

1.3. Briefly describe why “WHERE” clause is not used to select rows after a “GROUP BY” clause?

https://www.java67.com/2019/06/difference-between-where-and-having-in-sql.html

The main difference between WHERE and HAVING clause is that when used together with GROUP BY clause, WHERE is used to filter rows before grouping while HAVING is used to filter records after grouping.

Here is a comparison chart that differentiate between two clauses

Where Clause	**Having Clause**
The WHERE clause specifies the criteria which individual records must meet to be selected by a query. It can be used without the GROUP BY clause	The HAVING clause CANNOT be used without the GROUP BY clause
The WHERE clause selects rows before grouping	The HAVING clause selects rows after grouping
The WHERE clause cannot contain aggregate function	The HAVING clause can contain aggregate functions
The WHERE clause is used to impose condition on SELECT statement as well as single row function and is used before GROUP BY clause	The HAVING clause is used to impose condition on GROUP BY function; thus, it is used after GROUP BY clauses in the query
EXAMPLE: **SELECT Column, AVG(Column_name)** **FROM Table_name** **WHERE Column > Value** **GROUP BY Column_name**	Example: **SELECT Column, AVG(Column_name)** **FROM Table_name** **WHERE Column > Value** **GROUP BY Column_name** **HAVING Column_name > or < Value**

1.4. Briefly describe correlated subquery.

A correlated subquery (A.K.A., a synchronized subquery) is a subquery (a query nested inside another query) that uses values from the outer query.

We use correlated subquery because some data questions can only be answered with correlated subqueries.

https://www.geeksforgeeks.org/sql-correlated-subqueries/

EXAMPLE of Correlated Subqueries : Find all the employees who earn more than the average salary in their department.

```
SELECT last_name, salary, department_id
FROM employees outer
WHERE salary >
               (SELECT AVG(salary)
                FROM employees
                WHERE department_id =
                      outer.department_id);
```

1.5. Briefly describe the difference between identifying and non-identifying relationship.

An identifying relationship indicates that the **child table cannot be uniquely identified without the parent table**. It entails the instance when the existence of a row in a child table depends on a row in a parent table.

A non-identifying relationship is one where the **child table can be identified independently of the parent table**. It entails the instance when the primary key attributes of the parent *must not* become primary key attributes of the child.

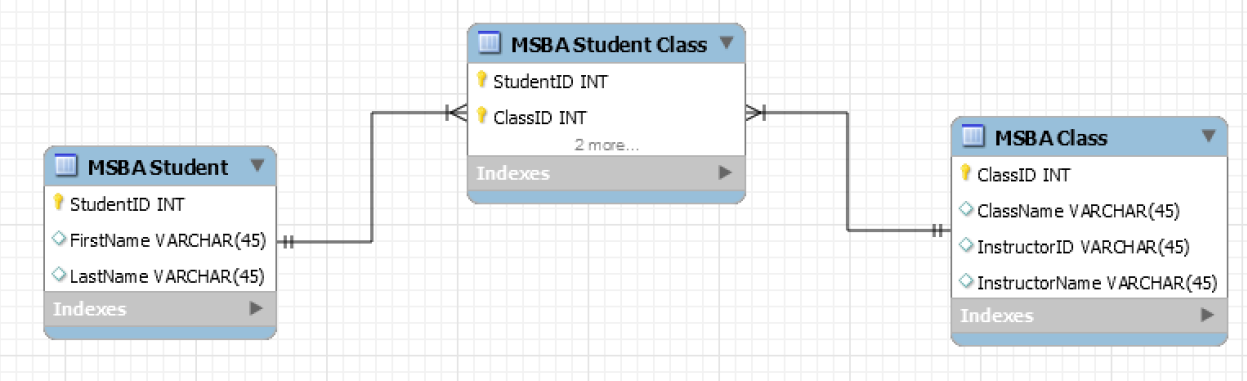
1.6. Briefly describe how you would create an ER diagram with M:N relationship between two entities (post an image from mySQL workbench)

https://www.relationaldbdesign.com/database-design/module6/convert-manyToManyRelationships-into-oneToManyRelationships.php

A *composite entity* (They are also known as an associative entity) links Entities in a many-to-many relationship in a special manner. A composite entity has only one role, which is to provide an indirect link between two entities in a M:N relationship.

A composite entity has also been coined a term, a linking table. It is noteworthy that a composite entity has no key attribute of its own; rather, it receives the key attributes from each of the two entities it connects and combines them together.

The following graphic illustrates a composite entity that now indirectly links the MSBA STUDENT and MSBA CLASS entities:



A composite entity called **MSBA STUDENT CLASSES** is created from a MSBA STUDENT entity and MSBA CLASS entity. Now notice that there exists M:N relationship between MSBA STUDENT and CLASS and it has been dissolved into two one-to-many relations:

1. There are one to many 1:N relationship between MSBA STUDENT and MSBA STUDENT CLASSES:
    1. for one instance of MSBA STUDENT, there exists 0, 1, or many instances of MSBA STUDENT CLASSES;
    2. however, for one instance of MSBA STUDENT CLASSES, there exists 0 or 1 instance of MSBA STUDENT.
2. The 1:N relationship between MSBA CLASS and MSBA STUDENT CLASSES reads this way:
    1. For one instance of MSBA CLASS, there exists 0, 1, or many instances of MSBA STUDENT CLASSES;
    2. but for one instance of MSBA STUDENT CLASSES, there exists 0 or 1 instance of MSBA CLASS.

2. Write SQL queries for the following cases (using sakila database):

2.1. (2 points) Check if a movie is in stock: list inventory ids that are currently in stock for movie “ACE GOLDFINGER”

```
# 2.1. Check if a movie is in stock: list inventory_ids that are currently in stock for movie “ACE GOLDFINGER”
SELECT inventory_id
FROM inventory
JOIN rental
USING (inventory_id)
JOIN film
ON inventory.film_id = film.film_id
WHERE title = "ACE GOLDFINGER" AND return_date > rental_date
GROUP BY inventory_id
ORDER BY inventory_id;
```

inventory_id
9
10
11

2.2. For movie in stock, checkout the movie (in rental table) for customer = “LINDA WILLIAMS” (assume your staff id = 1).

```
SELECT inventory_in_stock(9);

SELECT inventory_in_stock(10);

SELECT inventory_in_stock(11); # Inventory_in_stock is 1

SELECT customer_id
FROM customer
WHERE first_name = "LINDA" AND last_name = "WILLIAMS";

INSERT INTO rental(rental_date, inventory_id, customer_id, staff_id)
VALUES(NOW(), 11, 3, 1);
```

customer_id came out to be 3, which is inserted into rental table

No result shown after executing the INSERT statement.

2.3. For rental entered, collect the rental payment for the movie and add a row in the payment table.

```
SET @rentID = last_insert_id(), @balance = get_customer_balance(3, NOW());
INSERT INTO payment(customer_id, staff_id, rental_id, amount, payment_date)
VALUES(3, 1, @rentID, @balance, NOW());
```

No result shown (because values were inserted without having to return query value)

2.4. (2 points) Create a list of overdue movies (i.e. rental_duration < '2006-02-18' – rental_date).

```
SELECT film.title, customer.first_name, customer.last_name, rental.rental_date
FROM sakila.rental
INNER JOIN sakila.customer ON rental.customer_id = customer.customer_id
INNER JOIN sakila.inventory ON rental.inventory_id = inventory.inventory_id
INNER JOIN sakila.film ON inventory.film_id = film.film_id
WHERE return_date IS NULL AND rental_duration < DATEDIFF('2006-02-18', rental_date);
```

	title	first_name	last_name	rental_date
▶	PEACH INNOCENT	HEATHER	MORRIS	2006-02-14 15:16:03
	SUIT WALLS	ROLAND	SOUTH	2006-02-14 15:16:03
	SONS INTERVIEW	CATHY	SPENCER	2006-02-14 15:16:03
	TITANIC BOONDOCK	JUSTIN	NGO	2006-02-14 15:16:03
	TITANIC BOONDOCK	MORRIS	MCCARTER	2006-02-14 15:16:03
	LUST LOCK	MARGIE	WADE	2006-02-14 15:16:03

3. Designing Database: The athletics department at Emory University needs to create a database to track the games across all college sports as well as ticket sales. There are many college sports, including men’s basketball, women’s basketball, track and field, swimming, etc.

Each college sport has a corresponding university team. For each of these sports (teams), Emory maintains some basic information including team name, description, head coach, and year of establishment. There are also several facilities managed by Emory for these college sports, such as WoodPEC gymnasium, Cooper field, and Brown aquatic center. Each facility has a name, capacity, date of construction, location, and date of last inspection. Each game is between an Emory University team and another college team (e.g. Northwestern, Duke, Purdue).

Emory maintains attendance of each game (i.e. number of people attended the game), game date, final scores for the two teams, whether it is a home game, the facility used for the game, and type of the game (e.g. a conference or exhibition game). Customers can buy season tickets, or tickets for specific games. Season ticket prices are fixed for each sport. Each customer may buy one or more season tickets per sport.

Emory records season ticket sales in terms of who, when, which sport/team, and how many tickets has purchased. Similarly, Emory also tracks who bought game tickets, how many, for which game, and when. Unlike season tickets, however, game ticket prices may vary from transaction to transaction.

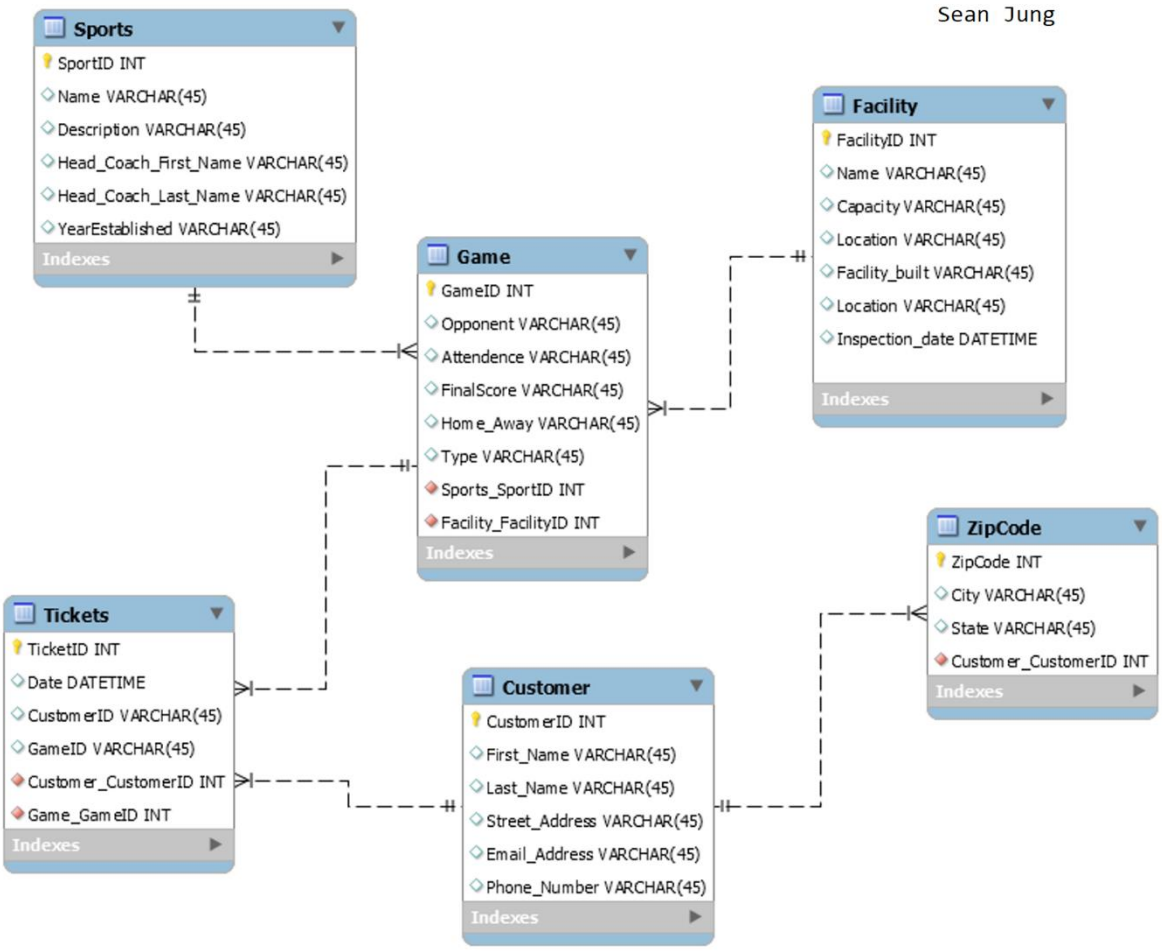
Hence, Emory also tracks how much a customer paid per ticket. Emory also keeps track of all customers who bought tickets, including their first name, last name, address (street_address, city, state, zip), email, and telephone.

3.1. Database Design (ER Model) Based on the above description, create an ER diagram. The requirements for the E-R diagram are as follows:

• The design should be in third normal form (3NF).

• The design should clearly identify all the primary and foreign keys.

• The design should clearly identify all the relationships. Specifically, the cardinality of the relationships should be clearly identified. No explicit many-to-many relationships are allowed. Draw your ER diagram and clearly list all your assumptions.



Assumption Made:

1. A game can only be played in one facility at time, meaning multiple games can't be played in one facility at the same time.
2. A customer may buy multiple game tickets (season tickets or multiple individual tickets), but each customer can only buy one game ticket, which is only valid for one game.

3.2. Dimensional Modeling The next step is to design a data warehouse about game ticket sales (you can exclude season ticket sales). The analytics team at Emory University hopes to conduct multi-dimensional analysis on game ticket sales to answer ad hoc questions such as:

• Which opponent generates the highest ticket sales for a particular sport?

➔ *Opponent* column under *Game* table help answer the question

• How far advance do customers buy tickets for a particular game?

➔ *Date* column under *GameTickets* table help answer the question

• How are sports ranked in terms of total game ticket sales?

➔ *Attendance* column under *Game* table help answer the question

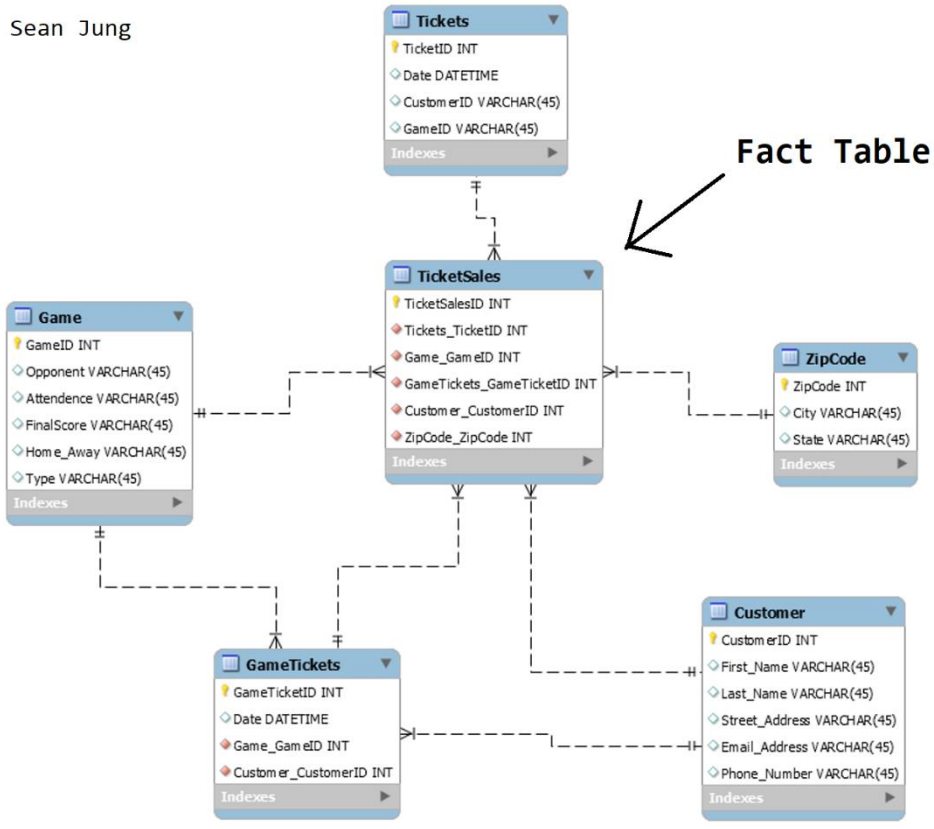
• How ticket sales vary by year, season, month, week, day of week, facility, customer city, zip code?

➔ It can be answered after joining different tables together

• How are sports ranked in terms of total ticket sales per game? Design a dimensional model with a single fact table (star schema) using MySQL workbench.

➔ It can be answered after joining different tables together

Sean Jung



Briefly discuss what is the grain of your fact table?

➔ Ticket sales by 1) Tickets 2) ZipCode, 3) Customer, 4) Game

4. TechCrunch reported that a startup Fivestar got another round of funding to grow their business. This startup seems to have created a “software to support small business payments and marketing”. Using this software, small business can “build a database of their own customers.” Assuming you are the CIO for the startup (Fivestar) that is growing rapidly and is now interested in migrating to big data technologies. **Identify what information a B2B company like Fivestar needs to store and discuss** (with reasons) **what NoSQL database(s) they should implement.**

Fivestar’s client need to find suitable way to store the data generated from their small business operations. The Type of data that need to be stored in the database varies depending on the nature of the business and its operation style; however, most small businesses regardless of its sector need to store its information about the customers (email/phone, physical address) and its product/service (cost, date and time of purchase/refund) which they offered to the customers.

And it is probably the case that they need to either constantly update whenever new customers are acquired (their contact information along with its identifier), existing customers went through transactions or new product is added to the catalog to appropriately reflect that on the database. Conversely, they need to somehow find ways to efficiently query the information in their database to extract certain information to generate insight that can acts as strategic guidance to their company’s positions on the marketplace.

Here, two types of NoSQL database are two close contenders: **1) Column-based or wide column NOSQL systems** or **2) Key-value stores**. Let’s first determine what each datastore is about.

Column-based or wide column NOSQL systems partition a table by column into column families (a form of vertical partitioning) where each column family is stored in its own files while also allowing versioning of data values. While column-based NoSQL has advantages of 1) being schema-free and therefore highly flexible, 2) allowing key-value pairs to be stored in a massively parallel system, 3) being more efficient for retrievals that only need to access some of the columns, 4) being common when multiple applications operate on top of the same data, and 5) allowing for better compression, this database system is inefficient 1) if used in OLTP with many row inserts and 2) if the data is small enough to run on a single server (because then a column family is unnecessary overkill).

On the other hand, key-value stores focus on high performance, availability, and scalability by storing data in a distributed storage system. The main key difference from the wide column NoSQL system is that the key is a unique identifier associated with a data item and is used to locate this data item rapidly. The value is the data item itself, and it can have very different formats for different key-value storage systems. The value can be stored in a variety of different formats such as records, objects, documents, and more complex format including Structured data rows (tuples) like relational data or Semi-structured data (using JSON). Thus, this database is optimized for making very quick reads and updates of entire items and operations such as Inserting (key, value), Fetching (key), Updating(key), Deleting (key). Advantages include that it is the simplest form of database management systems and it is incredibly fast (keys are short and easy to look up and efficient, fault-tolerant, great for applications that mainly require lookups.

Thus, each business need to somehow make tradeoff when choosing one of the two database system based on what they value more. Based on the advantages and disadvantages of the each of these two databases, I find that key-value based NoSQL data may be more appropriate. Here is the reason why: Compared to the large corporation, small business may not need to update their existing database often, but rather it weighs in more focus in generating insights from their data in efficient way.

Therefore, If I were the person who is responsible for making a call between the two, I would be most likely to suggest them use key-value based NoSQL format given that it is probably the best NoSQL database to perform such task aforementioned. In addition, Column-based NoSQL is not ideal to use when the data is small enough to run on a single server because then a column family is unnecessary overkill. Knowing key-value based pair is also optimized for storing session information, user profiles, user preferences, shopping cart data, for applications that have frequent small reads and writes and for applications that have simple data models, it further cements why key-value based pair should be preferred over the others in the context of the small business operations and also the fivestar’s database system that stores the all these repository database from each of their clients (small businesses).

5. In July 2020, Burger King used #CowsMenu on Twitter to present their company's commitment towards socially and environmentally responsible decisions.

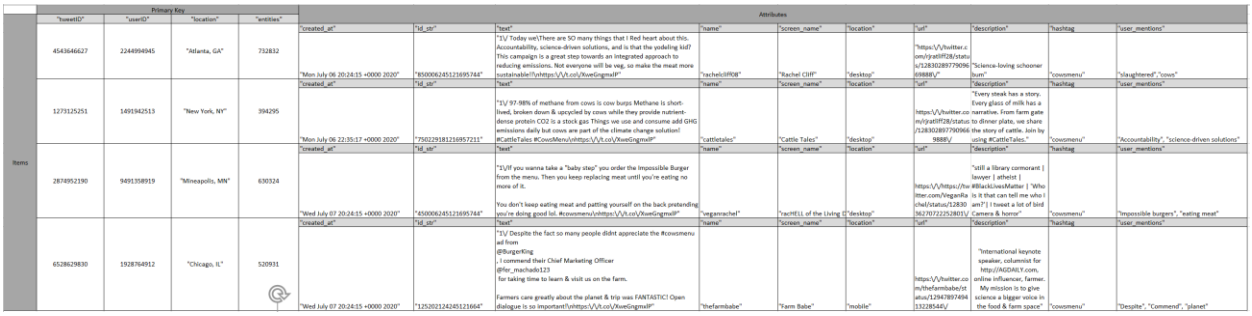
To achieve the success through social media marketing campaign, they engaged with their customer base on Twitter. Assuming Burger King saves every tweet that is related to them and their campaigns, briefly discuss what NoSQL data store will be most relevant for that Tweet (JSON) data.

5.1. Based on the tweet's JSON file and selected NoSQL data store, discuss the structure (schema) of the database.

https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/overview/intro-to-tweet-json (How Twitter's JSON Works)

https://www.alexdebrie.com/posts/dynamodb-single-table/ (Table Design in DynamoDB)

https://twitter.com/hashtag/cowsmenu?lang=en (Actual twitter's feed that is related to the hashtag "cowsmenu")



	Primary Key				Attributes									
	"tweetID"	"userID"	"location"	"entities"	"created_at"	"id_str"	"text"	"name"	"screen_name"	"location"	"url"	"description"	"hashtag"	"user_mentions"
Items	4543646627	2244994945	"Atlanta, GA"	732832	"Mon July 06 20:24:15 +0000 2020"	"850006245121695744"	"1\/ Today we\There are SO many things that I Red heart about this. Accountability, science-driven solutions, and is that the yodeling kid? This campaign is a great step towards an integrated approach to reducing emissions. Not everyone will be veg, so make the meat more sustainable!?\nhttps:\/\/t.co\/XweGngmxlP"	"rachelcliff08"	"Rachel Cliff"	"desktop"	"https:\/\/twitter.com/gratcliff28/status/1283028897790698887\/"	"Science-loving schooner bum"	"cowsmenu"	"slaughtered","cows"
	1273125251	1491942513	"New York, NY"	394295	"Mon July 06 22:35:17 +0000 2020"	"750229181216957211"	"1\/ 97-98% of methane from cows is cow burps Methane is short-lived, broken down & upcycled by cows while they provide nutrient-dense protein CO2 is a stock gas Things we use and consume add GHG emissions daily but cows are part of the climate change solution! #CattleTales #CowsMenu\nhttps:\/\/t.co\/XweGngmxlP"	"cattletales"	"Cattle Tales"	"desktop"	"https:\/\/twitter.com/gratcliff28/status/1283028897790669888\/"	"Every steak has a story. Every glass of milk has a narrative. From farm gate to dinner plate, we share the story of cattle. Join by using #CattleTales."	"cowsmenu"	"Accountability", "science-driven solutions"
	2874952190	9491358919	"Minneapolis, MN"	630324	"Wed July 07 20:24:15 +0000 2020"	"850006245121695744"	"1\/If you wanna take a "baby step" you order the Impossible Burger from the menu. Then you keep replacing meat until you're eating no more of it.\n\nYou don't keep eating meat and patting yourself on the back pretending you're doing good lol. #cowsmenu\nhttps:\/\/t.co\/XweGngmxlP"	"veganrachel"	"rachHELL of the Living D"	"desktop"	"https:\/\/twitter.com/VeganRachel/status/1283036270722252801\/"	"still a library cormorant	lawyer	atheist	#BlackLivesMatter	'Who is it that can tell me who I am'?	I tweet a lot of bird Camera & horror"	"cowsmenu"	"impossible burgers", "eating meat"
	6528629830	1928764912	"Chicago, IL"	520931	"Wed July 07 20:24:15 +0000 2020"	"1252021242451216647"	"1\/ Despite the fact so many people didnt appreciate the #cowsmenu ad from @BurgerKing , I commend their Chief Marketing Officer @fer_machado123 for taking time to learn & visit us on the farm.\n\nFarmers care greatly about the planet & trip was FANTASTIC! Open dialogue is so important!\nhttps:\/\/t.co\/XweGngmxlP"	"thefarmbabe"	"Farm Babe"	"mobile"	"https:\/\/twitter.com/thefarmbabe/status/1294789749413228544\/"	"International keynote speaker, columnist for http://AGDAILY.com, online influencer, farmer. My mission is to give science a bigger voice in the food & farm space"	"cowsmenu"	"Despite", "Commend", "planet"

Document-based NOSQL systems store data in the form of documents using well-known formats, such as JSON (JavaScript Object Notation). Documents are accessible via their document id but can also be accessed rapidly using other indexes. As you can see on the table above, there are four primary keys, each of which are connected to different attribute values. Based on the actual JSON formats used by Twitter's database system,

1) ***'tweetID'*** primary key leads to further information about its creation time of each tweet, its unique ID numbers, and its contents (text).
2) From ***'userID'*** primary key, it leads to information about the unique user ID, screen name, url of the tweet, and user account description.
3) From ***'location'***, user's physical location at which tweet has been made can be accessed.
4) Lastly, from the ***'entities'*** primary key, information about the hashtag, *'cowsmenu'* in this case, and user's mentioning of certain trigger words such as 'slaughtered' and 'accountability' can be archived.

```
  "tweetID": {
  "created_at": "Mon July 06 20:24:15 +0000 2020",
  "id_str": "850006245121695744",
  "text": "1\/ Today we\There are SO many things that I Red heart about this. Accountability, science-driven solutions, and
,
  "userID": {
    "id_string": "850006245121695744",
    "name": "rachelcliff08",
    "screen_name": "Rachel Cliff",
    "location": "desktop",
    "url": "https:\/\/twitter.com/rjratliff28/status/1283028977909669888\/
\/",
    "description": "Science-loving schooner bum"
},
  "location": {
      "location":"Atlanta, GA"
  },
  "entities": {
    "hashtags": ["cowsmenu"]
        },
    "user_mentions": ["slaughtered","cows"
    ]
  }
```


This is an example of what one instance looks like in terms of its JSON format.

5.2. Assuming Burger King decides to save the tweets in key-value document store (e.g. DynamoDB), write a syntax on how you would query the tweet (that mentions Burger King) creator's name. Please define the database schema first

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SQLtoNoSQL.CreateTable.html#SQLtoNoSQL.CreateTable.DynamoDB (Creating Schema)

https://stackoverflow.com/questions/60990793/dynamodb-scan-all-items-that-meet-condition (Querying the data)

Here below, I am creating a table named 'CowsMenu' that contains four primary

keys I specified previously under the question 5.1

```
// Create a table "CowsMenu" and its schemas

{
    TableName : "CowsMenu",
    KeySchema: [
        {
            AttributeName: "tweetID",
            KeyType: "HASH", //Partition key
        },
        {
            AttributeName: "userID",
            KeyType: "HASH", //Partition key
        },
        {
            AttributeName: "location",
            KeyType: "HASH", //Partition key
        },
        {
            AttributeName: "entities",
            KeyType: "RANGE" //Sort key
        }
    ],
    AttributeDefinitions: [
        {
            AttributeName: "tweetID",
            AttributeType: "S"
        },
        {
```

```
            AttributeName: "userID",
            AttributeType: "S"
        },
        {
            AttributeName: "location",
            AttributeType: "S"
        },
        {
            AttributeName: "entities",
            AttributeType: "S"
        }
    ],
    ProvisionedThroughput: {       // Only specified if using provisioned mode
        ReadCapacityUnits: 1,
        WriteCapacityUnits: 1
    }
}
```

Here below, I am querying all names of the tweets' creator who mentioned hashtag #cowsmenu.

```
// Query all names of the tweets' creators who mentioned hashtag "cowsmenu"


dynamodb = boto3.client('dynamodb')
paginator = dynamodb.get_paginator('scan')

response_iterator = paginator.paginate(
    TableName=CowsMenu,
    FilterExpression='hashtag = :filter',
    ExpressionAttributeValues={
        ":filter": {
            "S": "cowsmenu"
        }
    }
)

for name in response_iterator:
    print(name)
```