In the confines of mathematical set theory, as well as computer science database theory, normalization is a long developed concept of efficiency. In short, it is important to understand that normalization exists in database theory as a means of removing repeated data(redundancy) and reduce calls to those sets of data(dependency removal). Through a systematic process, large tables with lots of variables are condensed into more concise groupings, and these data tables are linked through common variables.

For the context of our classwork, SQL data normalization is all we care to go into detail with due to our use of mySQL in the SQL framework. Database theory currently has arguably around 6 different forms of normality that are widely used and accepted; however, for the sake of mySQL theory, we only care about and deal with the first three forms in the list. This paper will address the 1st, 2nd, and 3rd Normal forms, which will be colloquially addressed as 1NF, 2NF, and 3NF for the duration of this analysis. These are not the only useful normal forms, but as stated previously, these are the only true applicable and useful ones for mySQL databases.

1NF is the simplest normal form that we will address, but the other two forms build off of this normal form, so it is crucial to address this form first. 1NF requires only a basic understanding of set theory or even just an understands of non-repetitive data, as every single composite entry in a 1NF table has to be unique. If a table possesses 5 different categories/variables, for any given row, in 1NF notation, every other row is allowed to match no more than 4 categories. We bolster normalization on 1NF because duplicated data in the confines of storing information other than statistical data does not require duplicates. Redundancies only hurt our ability to gather and process data in almost every single case. Now, to understand I will provide an example of a table that follows 1NF and one that does not follow 1NF.

Name	DOB	Cash Transaction	Account Number	Current Day
Billy Bob	2002-02-02	100	11111	2018-03-23
Don Bob	2002-02-02	100	11111	2018-03-23
Janet Marge	1999-04-03	200	10101	2018-03-25
Don Quixote	1605-01-01	1	10000	2018-04-28

As can be seen from the table above representing something similar to a banking database, Billy Bob and Don Bob share everything in the database in common except for their names, and this is valid in 1NF, as will be shown in the next example, a counterexample, if the names were the same we would not be in 1NF notation. Every other entry in the table, Janet Marge and Don Quixote do not have any redundancy in their entries, so they are of no trouble.

Crayon Color	Sharpness	Length	Melting Potential	Circumference
Blue	0.57	0.25	0.43	0.67
Blue	0.57	0.25	0.43	0.67
Red	0.45	0.88	0.34	0.55
Green	0.34	0.31	0.66	0.33

This table provides a counterexample, or violation to the 1NF notation. As can easily be observed, the first two rows, Blue and Blue match perfectly with each of their elements. These

break the rule that all entries must be unique, and thus, this table does not fit the specifications of 1NF notation.

Before introducing 2NF, it is crucial to understand something used in both 2NF and 3NF which is the concept of the “key”. A key is a data variable or value that is capable of identifying a particular table row/entry/record uniquely. It can exist as a single column variable, commonly called a primary key, or it can be composed of multiple column variables, known as a composite key. Both types of keys must always possess a series of qualities. They can never be null for any given data entry. They must always be unique. They are immutable, meaning they cannot be changed.

2NF is much like it's predecessor, except it deals with the addition of one new requirement. All tables in 2NF must, as first contingency, be already existing in 1NF form. On top of this, the new condition for 2NF is that all tables in 2NF format must have a primary key associated with each data entry. Next we usually split our table into smaller tables with less information to enable quicker and less repetitive processing. Now, in order to link these two split up tables, we will assign every entry in the split table a key based on a particular column. Sometimes this key is an already existing column that we identify as the primary key, and sometimes this key is something we choose to generate. After generating all the primary keys for the first table, which follow the criteria above, we inject these primary keys to the corresponding rows in the second tables as what are often called foreign keys. Foreign keys can be, and often are duplicates as we will see in my provided example. Categorically, foreign keys can have a different name from their primary key, they do not have to be unique, that can be null, and their purpose, is to insure that the rows of one table correspond to the rows of another.

For some example: We take this table, table one

Name	Blood Type	Peanut Butter Preference	Political Affiliation
Bill	O	Crunchy	Democrat
Bill	A	Smooth	Republican
Bob	B+	Crunchy	Communist
Bob	A-	Crunchy	Socialist
Daryl	O+	Smooth	Green

And split it into two, assigning our primary keys based on name, if we are trying to find Peanut Butter preference, blood type and political affiliation information on all the people with the same first name.

Primary Key	Name
1	Bill
2	Bob
3	Daryl

Foreign Key	Peanut Butter Preference	Political Affiliation	Blood Type
1	Crunchy	Democrat	O
1	Smooth	Republican	A
2	Crunchy	Communist	B+
2	Crunchy	Socialist	A-
3	Smooth	Green	O+

A violation would exist if instead we chose our primary keys table like this, with keys representing two different data keys.

Primary Key	Name
1	Bill
2	Bob
3	Bob
4	Daryl

The final Normality type, 3NF requires the table to already exist in 2NF form, it also requires that there exist no transitive functional dependencies within a table. A transitive functional dependency occurs when changing anything other than the key may result in a change in something else in the table. If we have any transitive functional dependencies, we need to break a given table up further until there are no dependencies remaining. Let's start with an example in 2NF.

Song ID	Song Name
1	"Row Row Your Boat"
2	"Yeah!"
3	"The Metal"

SongID Foreign Key	Genre	Artist	Length

1	Lullaby	Unknown	0:45
2	Pop	Usher	3:14
3	Metal	Tenacious D	2:34

If we change the Song Title, Genre Artist and Length are all likely to change, we need to break these up so everything is based on the ID and there are no dependencies as shown below

Song ID	Song Title	GenreID
1	“Row Row Your Boat”	1
2	“Yeah!”	2
3	“The Metal”	3

GenreID	Genre	ArtistID
1	Lullaby	1
2	Pop	2
3	Metal	3

ArtistID	Artist	LengthID
1	Unknown	1
2	Usher	2
3	Tenacious D	3

LengthID	Length
1	0:45

2	3:14
3	2:34

Now, only changing the SongID will affect all other data, everything else is transitively independent.

SongID Foreign Key	Genre	Artist	Length
1	Lullaby	Unknown	0:45
2	Pop	Usher	3:14
3	Metal	Tenacious D	2:34

The table is 2NF has transitive dependencies, making it a violation.

Thus concludes a study in normality.