**Pipeline with SQL**

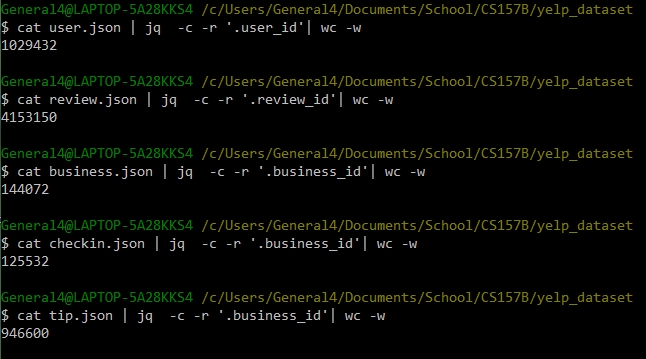
Note to professor - some of this information that follows is from my homework 1. I added to and/or updated information as necessary to better reflect the efforts I undertook during the MySQL pipeline exercise. Actually, the amount of new information far out weighs the amount from the previous homework.

**Extract and Transform**

The Yelp dataset JSON and CSV files were given to the class to use for this assignment. Instead of working with the JSON files, I initially decided to work with the csv files. These were files that were converted to csv from JSON. I knew better than to expect these files to be clean and well-formed so I inspected them before I tried to get them into MySQL. Some of the files I couldn’t open with my go-to text editor, NotePad++. So I decided to try the CSVKit command-line tool kit. Since I never used it before, I was wasting a lot of valuable time just trying to learn to use it. With work and other responsibilities in mind, I decided to abandon using CSVkit and try a GUI csv file reader. I used CSVed which was able to open most of the csv files. The first file I opened and inspected was tip.csv. As suspected it contained dirty data. There were some rows of data that contained entries that had many commas. Other entries were missing data from columns. I believe that since a comma was used as a delimiter during the conversion process, entries that contained commas caused incorrect rows to be entered. Since the csv files seemed to be so problematic, I decided to work with the JSON data. I used a Java program that parsed the data and used pipe symbols as delimiters. The resulting files were triple pipe separated files.

Before I could load the transformed data, I needed to count the number of records in each JSON file. I used command line tools for this. The initial inspection of the JSON files showed that each record was on it’s own line, I simple selected an id from each dataset and coundted the number of lines.

The following image shows the count results.



**LOAD**

I created a SQL schema based on the JSON object data of the dataset that is provided when you download the Yelp dataset. I was able to infer the data types by reviewing this data and by reviewing a few lines of each JSON file. Most of the id’s where mixed characters and not integers so they were assigned the VARCHAR type. The information showed that almost all the datasets had a unique id, for instance the Business set had a business\_id and the User set had a user\_id field. In my schema design I made these fields primary keys so that MySQL can use these as indexes. The review dataset contained multiple id fields – review\_id, user\_id, business\_id. Since the review\_id field was unique I used this as a primary key and made the other two id’s foreign keys since they referenced primary keys in other tables. For this initial loading of the data I did not want to impose any further constraints on the data. The following is a sample of the schema that shows the meta-data of the review schema.

**DROP** **TABLE** **IF** **EXISTS** REVIEW**;**

**CREATE** **TABLE** REVIEW

**(**

review\_id **VARCHAR(**30**),**

user\_id **VARCHAR(**30**)** **REFERENCES** **USER(**user\_id**),**

business\_id **VARCHAR(**30**)** **REFERENCES** BUSINESS**(**business\_id**),**

stars **INT,**

**date** **DATE,**

text **VARCHAR(**5010**),**

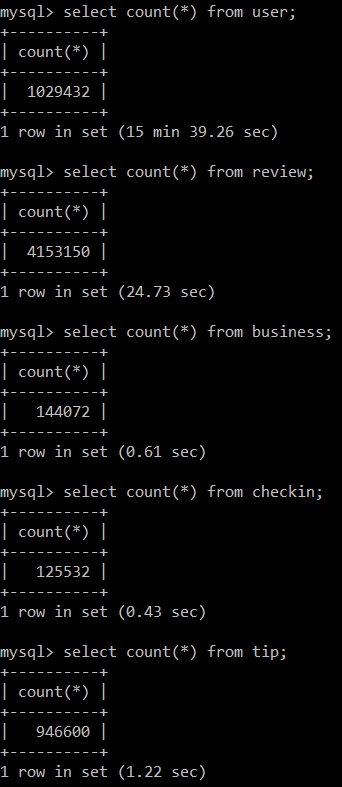
useful **INT,**

funny **INT,**

cool **INT,**

**type** **VARCHAR(**10**)**

**);**

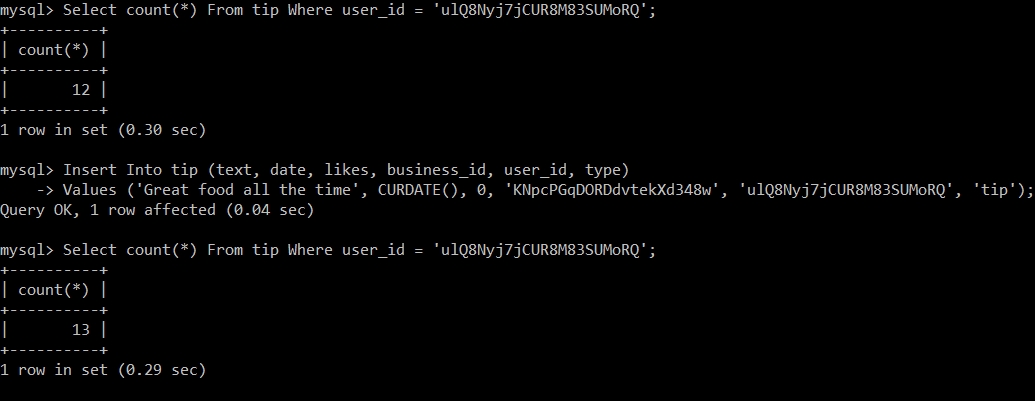


The SQL file that builds the schema also populates the tables with data. It uses the LOAD DATA LOCAL INFILE ‘<FILE-NAME>’ command. To inform MySQL that the data is delimited by triple pipe symbols, the script includes the command, COULUMNS TERMINATED BY ‘|||’. I ran the script using the MySQL command-line. It took well over an hour to load the data into the dataset! I started to think something went wrong, but it finished and had no errors. Next, I needed to check that all the data was loaded, so I counted the records in each set and checked to see that they matched the counts from the JSON files – it did. Successful import, the counts matched the JSON file counts. Note that it took MySQL over fifteen minutes to count the rows of the user table due to the number of records it contains.

**Query**

**Query #1 OTLP**

For my first query, I chose to do a simple insert. I selected at random a user\_id and business\_id to use for the insert. The image below shows the count for this user\_id before the insert and then again after the insert.



Notice that the insert operation is much faster than each Count operation. MySQL must count each record in the tip table for the Count operation and only write the data to the DB for the insert operation. For another slightly more complicated query I counted the distinct names in the user table. Since the user table contains over a million users, it took a long time, over fifteen minutes, for MySQL to count the distinct names.

**Query #2 (Simple Query)**

**SELECT** **COUNT(DISTINCT** name**)** **FROM** **user;**

+----------------------+

| COUNT(DISTINCT name) |

+----------------------+

| 88722 |

+----------------------+

1 row in set (15 min 29.80 sec)

The time it took for this query to run would be unacceptable for an on-demand or streaming web service. If this information was needed in such an app, it would need to be previously obtained and stored in a materialized view. Then the app could query the view and obtain the data much faster.

Next, I ran a more complex query that involved the join of two tables and the aggregation function Count.

**Query #3 (OLAP)**

**SELECT** business**.**business\_id**,** name**,** **COUNT(**review**.**stars**)** **as** num\_reviews

**FROM** business **JOIN** review **ON** business**.**business\_id **=** review**.**business\_id

**GROUP** **BY** business\_id **Having** num\_reviews **>** 1000 **ORDER** **BY** num\_reviews**;**

+------------------------+---------------------------------------------+-------------+

| business\_id | name | num\_reviews |

+------------------------+---------------------------------------------+-------------+

| sNVGdeOPeitJ3OWUQBINzQ | BabyStacks Cafe | 1002 |

| IMLrj2klosTFvPRLv56cng | Honey Salt | 1005 |

| 4k3RlMAMd46DZ\_JyZU0lMg | Ramen Sora | 1008 |

| IsoLzudHC50oJLiEWpwV-w | Chelsea's Kitchen | 1012 |

| gTlDDzDEHyDQ6iwjNhpI6A | Mount Everest India's Cuisine | 1016 |

| 0NmTwqYEQiKErDv4a55obg | Scarpetta | 1028 |

| OARQDsxyoGnnX2FfSl9HjA | Le Reve - The Dream | 1031 |

| q-zZgXKAQFLEgMp9ZNllgQ | Umiya Sushi | 1033 |

| NCFwm2-TDb-oBQ2medmYDg | Fountains of Bellagio | 1035 |

| 3N9U549Zse8UP-MwKZAjAQ | Liberty Market | 1036 |

| PVTfzxu7of57zo1jZwEzkg | FEZ | 1039 |

| EnCIojgP5KTr1leaysFE3A | Viva Las Arepas | 1045 |

| UUGoM4q4i8rK2CBRS0xDAw | Jean Philippe Patisserie | 1046 |

...

| Wxxvi3LZbHNIDwJ-ZimtnA | The Venetian Las Vegas | 2780 |

| MpmFFw0GE\_2iRFPdsRpJbA | XS Nightclub | 2828 |

| g8OnV26ywJlZpezdBnOWUQ | Hash House A Go Go | 2835 |

| FaHADZARwnY4yvlvpnsfGA | McCarran International Airport | 2865 |

| El4FC8jcawUVgw\_0EIcbaQ | MGM Grand Hotel | 3016 |

| AV6weBrZFFBfRGCbcRGO4g | Luxor Hotel and Casino Las Vegas | 3166 |

| iCQpiavjjPzJ5\_3gPD5Ebg | Secret Pizza | 3258 |

| rcaPajgKOJC2vo\_l3xa42A | Bouchon at the Venezia Tower | 3292 |

| SMPbvZLSMMb7KU76YNYMGg | ARIA Resort & Casino | 3385 |

| 5LNZ67Yw9RD6nf4\_UhXOjw | The Cosmopolitan of Las Vegas | 3400 |

| KskYqH1Bi7Z\_61pH6Om8pg | Lotus of Siam | 3458 |

| ujHiaprwCQ5ewziu0Vi9rw | The Buffet at Bellagio | 3482 |

| 2weQS-RnoOBhb1KsHKyoSQ | The Buffet | 3676 |

| f4x1YBxkLrZg652xt2KR5g | Hash House A Go Go | 3881 |

| eoHdUeQDNgQ6WYEnP2aiRw | Serendipity 3 | 3911 |

| hihud--QRriCYZw1zZvW4g | Gangnam Asian BBQ Dining | 4120 |

| DkYS3arLOhA8si5uUEmHOw | Earl of Sandwich | 4655 |

| cYwJA2A6I12KNkm2rtXd5g | Gordon Ramsay BurGR | 5116 |

| K7lWdNUhCbcnEvI0NhGewg | Wicked Spoon | 5216 |

| RESDUcs7fIiihp38-d6\_6g | Bacchanal Buffet | 5715 |

| 4JNXUYY8wbaaDmk3BPzlWw | Mon Ami Gabi | 6414 |

+------------------------+---------------------------------------------+-------------+

177 rows in set (1 min 8.56 sec)

The long running time of this query( > 1 minute) is likely directly proportional to the dataset size. It had to join two tables that contain a lot of data and then perform a mathematical operation on this data. If this query involved the user table it would have taken much longer since it contains millions of records. Even still, a query

that takes over a minute in a real-world application would not be good. This type of query should be chached if this data is needed often.

In Query #4 I joined the user and review tables and the result took much longer. I believe that one reason for the additional time is that I joined the review table with the user table which is much larger than the business table of the first query. The result was the empty set. Over an hour just to get the empty set!

**Query #4**

**SELECT** review**.**user\_id**,** name

**FROM** **user** **NATURAL** **JOIN** review **WHERE** review**.**text **LIKE** '%great%'**;**

Empty set (1 hour 29 min 24.24 sec)

When SQL has to join multiple tables with millions of records it is ridiculously slow. What made this query even slower was the fact that it had to search for something of which it had no index for. MySQL literally had to comb through every text entry of the join looking for the word ‘great’. When MySQL does not have and index to help with searches, it has no choice but to check record-by-record and word-by-word.

One thing I really like about SQL is how a query can include multiple subqueries. You can break a problem into multiple sub-problems and use the results of each sub-problem to solve the greater problem. The following query is a simple example that demonstrates this. It finds the users that have an average\_star value that is greater than the average of all the users average\_star values (names contained in the query result are not shown since there are over half million names in the result set).

**Query #5 (OLAP)**

**SELECT** name **FROM** **user** **WHERE** average\_stars **>** **(SELECT** **avg(**average\_stars**)** **FROM** **user);**

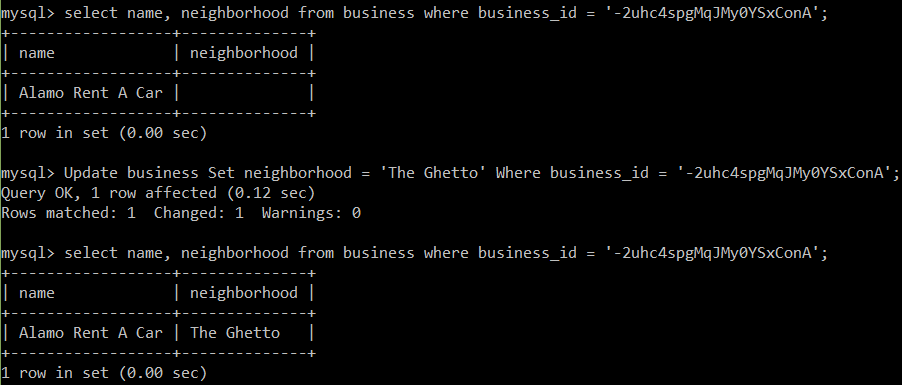
...

+----------------------------------+

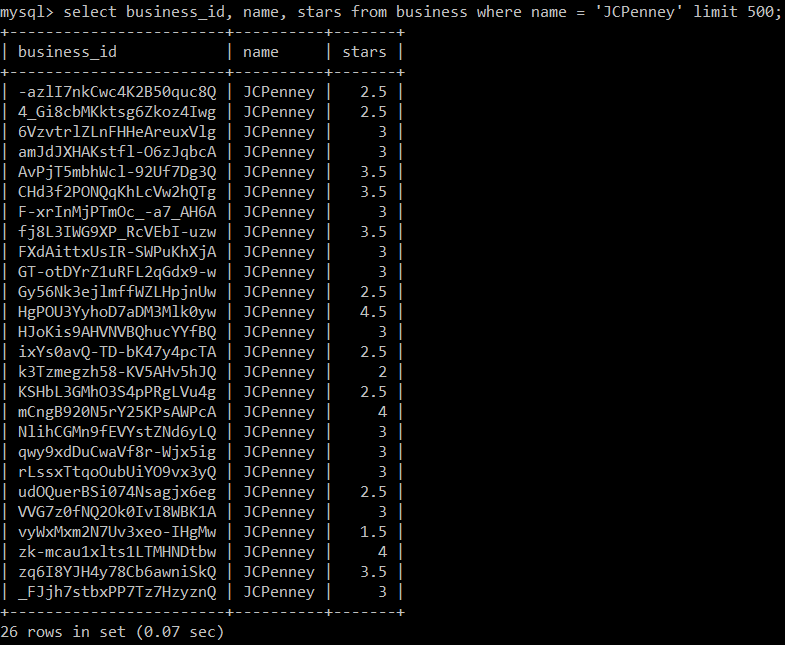
583834 rows in set (21 min 45.07 sec)

**Update**

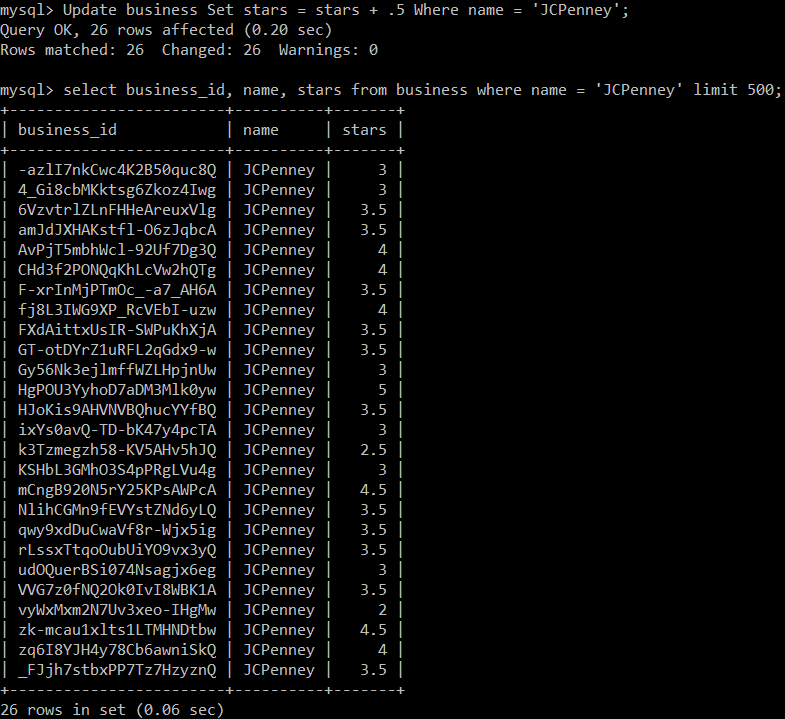
For a simple update, I choose to update a business that had an empty string for it’s neighborhood value and give it the value ‘The Ghetto’. The update took 0.12 seconds which is slightly longer than the 0.04 seconds of query #1 which was an insertion. I believe this is due to the fact that an insertion is faster since the database has a pointer to the last element and simply writes to the next location. In contrast, an update must use it’s index table to locate the item to be updated and then it can write to the location.



Next, I did a batch update. I updated the star value by .5 for every business named JCPenny in the business table. The images that follow show the star count before and after.



The JCPenny star count before the batch update



The JCPenny star count after the batch update

The batch update took longer than the single-row update which is to be expected since MySQL needed to update many records.

**Optimize Quality**

One way to improve my schema design that would help preserve data quality would be to add default values and attribute based constraints. For example, the review table has many fields that have a type of INT. It makes sense to give these fields an initial value of zero if there is no value inserted and since there is a default value it makes sense to add a NOT NULL check for each row. Since I didn’t want to re-load the data, I opted to Alter the existing table schema using the following command.

mysql**>** **ALTER** **TABLE** review **MODIFY** stars **INT** **NOT** **NULL** **DEFAULT** 0**;**

Query OK**,** 0 **rows** affected **(**18 **min** 38.60 sec**)**

Records**:** 0 Duplicates**:** 0 Warnings**:** 0

Since the star value is supposed to be an integer between zero and five, it makes sense to provide a range check for the values that try to be inserted. The SQL standard provides an attribute CHECK option that MySQL supports, by accepting it and ignoring it. So if you do the following MySQL will ignore it:

**ALTER TABLE review**

**ADD CHECK (stars < 5) 🡨 MySQL allows this but ignores it!**

Therefore, it was necessary to implement a trigger to check the value being assigned to the star attribute. I made a SQL file named star\_check.sql with the following script. This trigger will reinforce data integrity by ensuring that if validation fails at the code level, the database level will stop the star value from becoming corrupt.

**DROP** **TRIGGER** **IF** **EXISTS** checkStarsTrigger**;**

delimiter **//**

**CREATE** **TRIGGER** checkStarsTrigger

**BEFORE** **INSERT** **ON** review

**FOR** **EACH** **ROW**

**BEGIN**

**IF** **NEW.**stars **>** 5 **or** **NEW.**stars **<** 0 **THEN**

**SET** **NEW.**stars **=** 0**;**

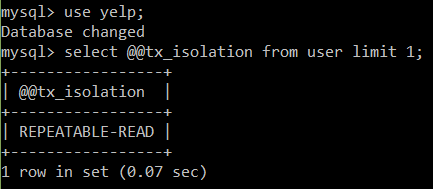
**END** **IF;**

**END;//**

delimiter \**;**

Similar other constraints can be added to the Yelp dataset to help preserve data integrity.

MySQL is very good at producing reliable results even with many concurrent operations being performed on a dataset. MySQL version 5.6, the version I am using on my machine, uses the InnoDB storage engine by default. The default transaction level for InnoDB is Repeatable-Read. I checked this by checking the isolation level of the user table.



With this isolation level MySQL uses row locks and gap locks to guarantee accurate information with select queries. If two different transactions query the same rows that contain gaps they will both get the same result. The first transaction locks the rows and any gaps between rows, but since the second transaction is only selecting rows and not updating or inserting, it is passed the same rows as the first transaction. However, if the second transaction tries to do an insert or update an error will occur (ERROR 1205 (HY:000) : Lock wait timeout exceeded …”). This is to prevent phantom reads and guarantee consistent results for the first transaction that locked the rows. If the first transaction does another select query, it will get the same results as before. The second transaction will only be able to write to the locked rows after the first transaction is finished (does a commit). This type of concurrency control ensures reliable data at a cost of reducing parallelism slightly.

**Optimize Performance**

**Materialized View Example**

Materialized views store the results of long-running queries in a table. Then, to get the results of the long-running query very fast, you can query the view table. I decided to try and optimize the query #3 of this paper which took over a minute by using a materialized view. One reason query #3 took so long is because it had to join the business and review table since the business name was needed in the resultset. I created a SQL script that creates a table that will hold the captured data. This script also runs the query that populates the materialized view. The materialized view will contain businesses with greater than 1000 reviews. The file business\_1k\_reviews.sql does the work.

**CREATE** **TABLE** Greater\_than\_1k**(**

business\_id **VARCHAR(**128**)** **NOT** **NULL,**

name **VARCHAR(**128**)** **NOT** **NULL,**

num\_reviews **INT** **NOT** **NULL,**

**UNIQUE** **INDEX** business\_mv **(**business\_id**)**

**);**

**INSERT** **INTO** Greater\_than\_1k

**SELECT** business**.**business\_id**,** name**,** **COUNT(**review**.**stars**)** **as** num\_reviews

**FROM** business **JOIN** review **ON** business**.**business\_id **=** review**.**business\_id

**GROUP** **BY** business\_id **Having** num\_reviews **>** 1000 **ORDER** **BY** num\_reviews**;**

mysql> source business\_1k\_reviews.sql;

Query OK, 0 rows affected (0.36 sec)

Query OK, 177 rows affected (1 min 12.08 sec)

Records: 177 Duplicates: 0 Warnings: 0

As you can see, to populate the materialized view took very close to the time to run Query #3 which is to be expected. However, the time to view the materialized view’s data was extremely fast – under a second!

**Below is the query that selects data from the materialized view table. The result set was produced fast!**

mysql> **SELECT** **\*** **FROM** Greater\_than\_1k**;**

+------------------------+---------------------------------------------+-------------+

| business\_id | name | num\_reviews |

+------------------------+---------------------------------------------+-------------+

| --9e1ONYQuAa-CB\_Rrw7Tw | Delmonico Steakhouse | 1311 |

| -ed0Yc9on37RoIoG2ZgxBA | Le Thai | 1117 |

| 0FUtlsQrJI7LhqDPxLumEw | Joe's Farm Grill | 1410 |

| 0NmTwqYEQiKErDv4a55obg | Scarpetta | 1028 |

| 0W4lkclzZThpx3V65bVgig | Schwartz's | 1603 |

| 2iTsRqUsPGRH1li1WVRvKQ | Carson Kitchen | 1225 |

| 2weQS-RnoOBhb1KsHKyoSQ | The Buffet | 3676 |

...

| YPavuOh2XsnRbLfl0DH2lQ | Lo-Lo's Chicken & Waffles | 1276 |

| yQab5dxZzgBLTEHCw9V7\_w | Charlotte Douglas International Airport | 1385 |

| z6-reuC5BYf\_Rth9gMBfgQ | St. Francis Restaurant | 1195 |

| zdE82PiD6wquvjYLyhOJNA | KoMex Fusion | 1128 |

| ZjSzUWHtnpCfjsa7CksSOg | The Signature at MGM Grand | 1356 |

| ZkGDCVKSdf8m76cnnalL-A | Le Village Buffet | 1864 |

| \_w5hBpkjHs5\_Hv3pLeHtIw | Julian Serrano | 1429 |

+------------------------+---------------------------------------------+-------------+

177 rows in set (0.00 sec)

The above query shows that using materialized views can greatly speed up data retrieval of queries that normally run slow. The down-side is that you must update materialized views periodically.

Query #3 time without optimization (1 min 8.56 sec)

Query #3 with optimization (0.00 sec)

The rate at which you update the materialized views depends on how the database is being used. For instance, a company that is analyzing the data may only need daily updates. However, an on-line business may require hourly updates so their view data is a closer reflection real time data.

**Vertical Partitioning Example**

Another optimization technique for large datasets is to use vertical partitioning – divide the table vertically by putting the columns that are used frequently into another table. This second table will speed up queries since there will be less disk scans/reads. The business table has a column named ‘attributes’. This coulumn is an array of the business’s attributes – BusinessAcceptsCreditCards: TRUE, BikeParking: FALSE, … I used a command-line tool called **jq** to extract the business\_id and attribute array from the business.JSON file and store them in a file named business\_attributes.json.

cat business.json | jq -c -r '.business\_id','.attributes' > business\_attributes.json

Here are two of the businesses from business\_attributes.json:

cdk-qqJ71q6P7TJTww\_DSA

["BusinessAcceptsCreditCards: True","RestaurantsPriceRange2: 2","WiFi: free"]

Q9rsaUiQ-A3NdEAloy0aJA

["BusinessAcceptsCreditCards: True","ByAppointmentOnly: True","RestaurantsPriceRange2: 3"]

I then wrote a python script that parses this data by cleaning, filtering and extracting each business and their corresponding attributes and stores them in a CSV file.

The first coulumn is the business id and the seven binary numbers that follow represent seven of the attributes. Here is what each column represents in-order - **business\_id, BikeParking, BusinessAcceptsCreditCards, GoodForKids, HasTV, OutdoorSeating, RestaurantsDelivery, WheelchairAccessible**

0DI8Dt2PJp07XkVvIElIcQ,1,1,0,0,0,0,1

LTlCaCGZE14GuaUXUGbamg,0,1,0,0,0,0,0

EDqCEAGXVGCH4FJXgqtjqg,1,1,1,1,0,1,0

cnGIivYRLxpF7tBVR\_JwWA,0,1,0,0,0,0,0

I then wrote a SQL script to build the table these attibutes would be stored in and populate the table.

**DROP** **TABLE** **IF** **EXISTS** BUSINESS\_ATTRIBUTES**;**

**CREATE** **TABLE** BUSINESS\_ATTRIBUTES

**(**

business\_id **VARCHAR(**30**)** **REFERENCES** BUSINESS**(**business\_id**),**

bike\_Parking **BOOLEAN,**

accepts\_credit\_cards **BOOLEAN,**

good\_for\_kids **BOOLEAN,**

has\_tv **BOOLEAN,**

outdoor\_seating **BOOLEAN,**

restaurants\_delivery **BOOLEAN,**

wheelchair\_accessible **BOOLEAN,**

**PRIMARY** **Key(**business\_id**)**

**);**

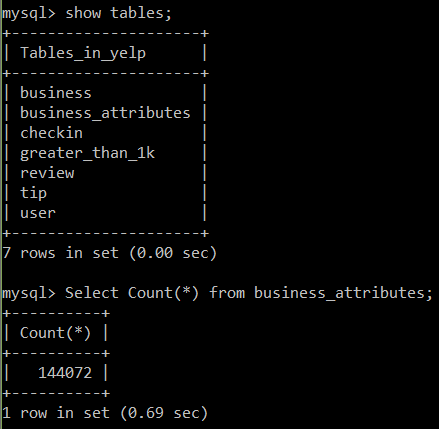
LOAD **DATA** **LOCAL** INFILE 'C:\\Users\\General4\\Documents\\School\\CS157B\\yelp\_data\\businessAttributes.csv'

**INTO** **TABLE** business\_attributes

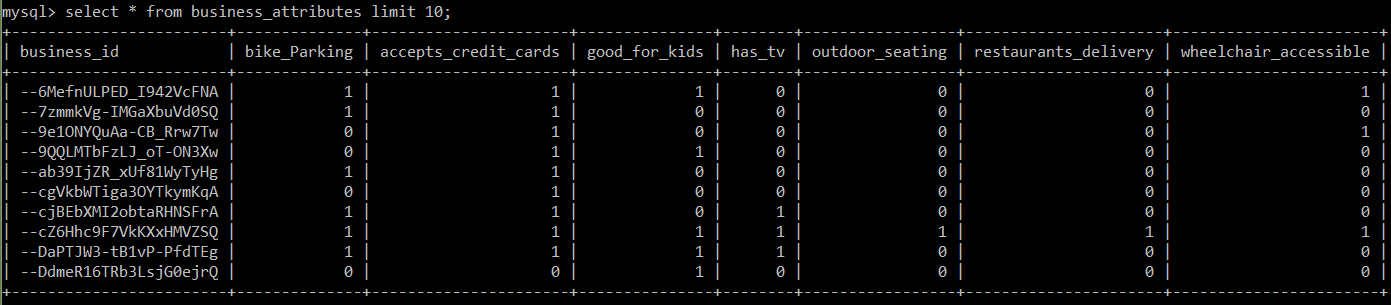
COLUMNS TERMINATED **BY** ','

LINES TERMINATED **BY** '\n'**;**

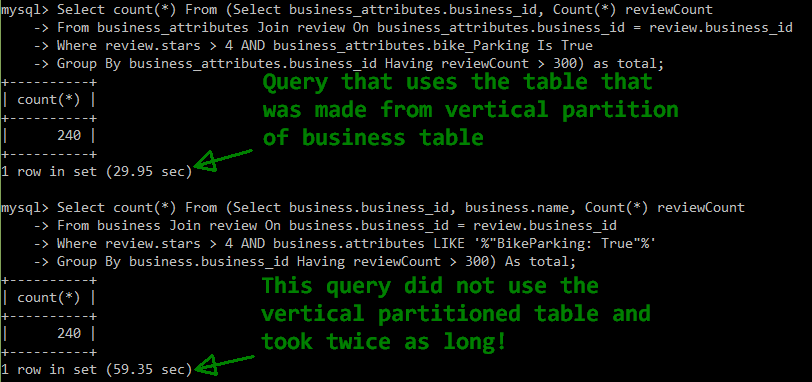
A quick check to see if the row count of business\_attributes matched the row count of business told me that all script was successful.



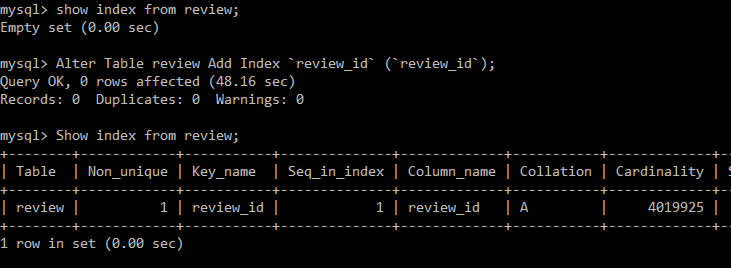
The following is a screenshot showing 10 of the rows of business\_attributes table (sorryI can’t make the image larger an still fit correctly on the page).

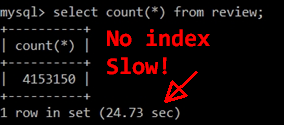


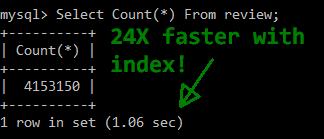
To test the effect of vertical partitioning of the business table I ran one Join query that used the table that was created from the partitioned data (attributes data of business tables) and ran one query that used just the full business table. The query finds the number of businesses that have greater than 300 reviews and have a BikeParking attribute that is true. As you can see from the screen shot, the query that did not use the partitioned data took twice as long! This shows how valuable vertical partitioning can be.



Similar performance gains can be obtained by using a horizontal partition as well. I could select all the businesses that have greater than 5 stars and put them into a separate table. Then queries that used this data would run much faster. I suspect the temporal gains would be similar to that of using vertical partitioning. Adding an index will also help speed up search queries. Initally, I did not give the review table an index to facilitate importing the data quickly – the less constraints imposed the quicker the data loading. I altered the review table by making the review\_id an index.





As the screenshots to the left show, adding an index can greatly improve select queries! This concludes my MySQL pipeline report. I just want to say that I learned a lot from this lesson. It helped solidify much of what I learned in CS157A and gave me a better understanding of relational databases and how big data works on this type of database.