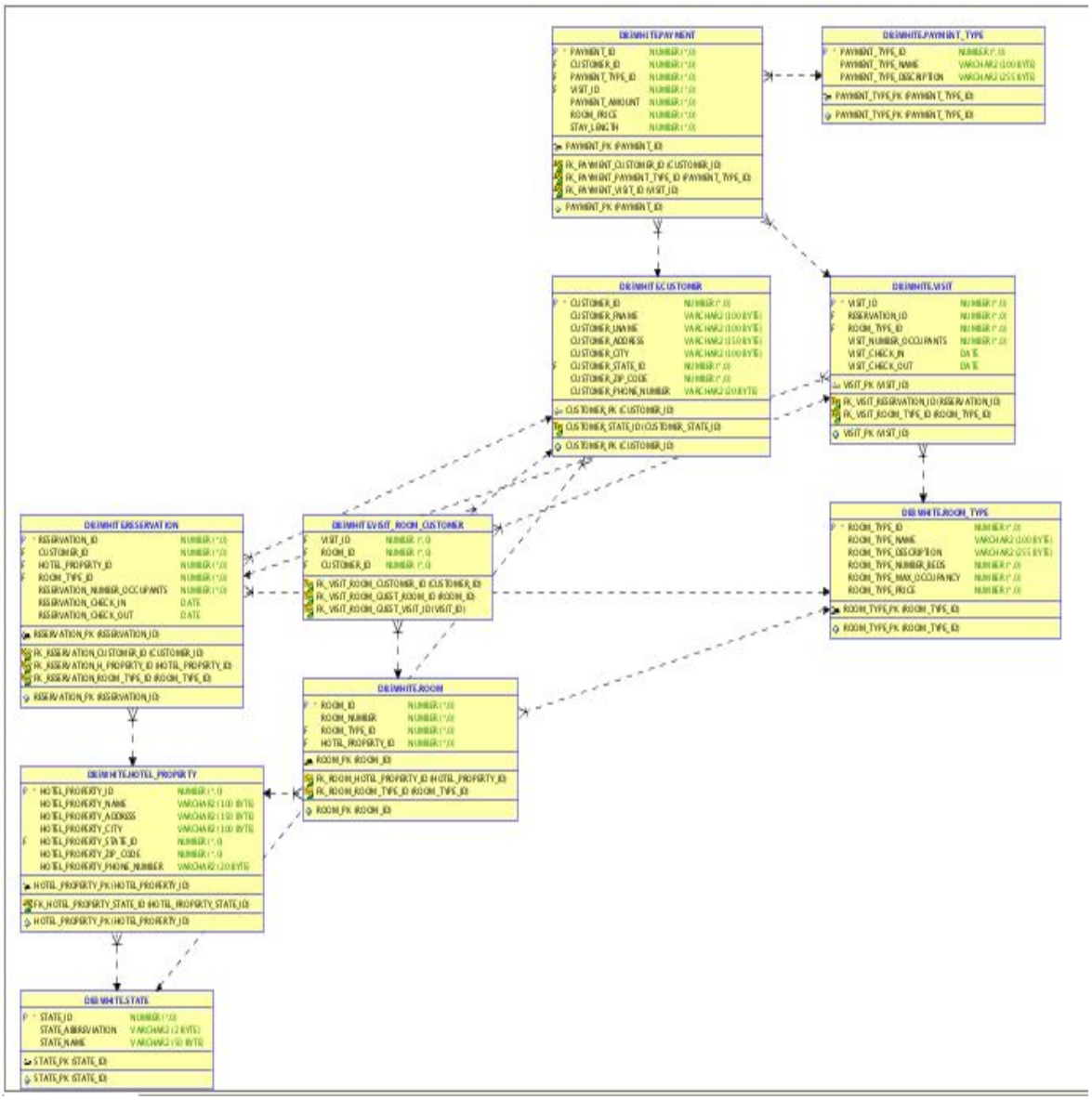
Final Project

This database was created to support a guest tracking system for a hotel chain. It allows the owners of a hotel to enter the rooms available at their hotel, along with the number of beds and maximum guest capacity, so that anyone looking for a hotel to stay in at a particular location can easily find vacant rooms that fit their needs. Reservations are tracked by customer, along with when they checked in and out, and how they paid for their stay. Using the data from the database, it will be possible to check various types of statistics, like what types of room are the most popular, which hotels have guests staying there the longest, and what times of the year hotels in a given area are booked the most. There are also a number of privacy features, like purging reservations after a given number of days, and clearing out guest data for guests who haven't booked a room in a while.

Topic Area	**Group Member**	**Points**
Database Design	William	25
Query Writing	Ritik	25
Performance Tuning	Srivatsan and Ankur	25
Other Topics	Mary	25

Database Design



Below is an explanation of the entity resource diagram, its tables and fields:

- The `HOTEL_PROPERTY` table contains information about each hotel property that is tracked by this database. It has the name, address, phone number, and an ID to identify the property.
- The `STATE` table contains the full and abbreviated name for all of the states. This is used with the `HOTEL_PROPERTY` table, as well as the `CUSTOMER` table, for addresses.
- The `ROOM` table contains information about each room in each hotel property. It has the room type, room number, hotel property, and an ID to identify the room.
- The `ROOM_TYPE` table contains information about the different types of rooms available. It has a name for the room, description of the room, number of beds, maximum guest occupancy, price per night, and an ID to identify the room type.

- The CUSTOMER table contains information about each customer that has made a reservation at one of the hotels. It contains their name, address, phone number, and an ID to identify them.
- The RESERVATION table contains information about the reservations made by customers. It contains the number of occupants, check in and check out dates, and IDs to identify the reservation, customer, room, and hotel property.
- The VISIT table contains information about the times when a customer stays at the hotel. It's largely similar to the RESERVATION table, and could probably be integrated into that table with some modification to avoid data redundancy. The purpose of this table is to track when the customer actually checked in and out to gather statistics about timing.
- The VISIT_ROOM_CUSTOMER table links a VISIT to a CUSTOMER and a ROOM.
- The PAYMENT and PAYMENT_TYPE tables store information about customer payments and what method they used to pay.

The database was created using the DB3WHITE group account.

While creating the database, we ran into a particularly annoying issue. We could create tables, but couldn't insert data into those tables, nor could we make indexes on those tables. As it turns out, the group account we were using had a default tablespace of COLORS, but had no quota set on that tablespace, making it impossible to insert data into tables or create indexes. It did have a quota set for the STUDENTS tablespace, but the account didn't have the privileges to change its own default tablespace to use that one instead. The individual student accounts had a default tablespace of STUDENTS, so we were able to create tables and insert data on our own accounts. This issue was resolved by specifying the STUDENTS tablespace in all creation queries.

Once overcoming the tablespace issue, we set out to fill our tables with relevant data at a scale that was large enough to allow us to explore queries and optimization in near real world conditions for such a database. In order to fill the database properly, we needed to start with the entities that do not rely on constraints from other tables, but that other tables depend on for foreign keys. This meant starting with CUSTOMER and HOTEL_PROPERTY. The hotel properties were generated from a mix of two different approaches. One was a Python script to make requests to this address generator site (https://www.fakeaddressgenerator.com/World/us_address_generator) and parse out the various parts of the address. This method also used a hotel name generator (https://www.fantasynamegenerators.com/hotel-names.php) that created some interesting names for these hotels. The second approach was to gather some names of real hotels and combine those with random addresses to supplement the amount of names generated in the first method. Customers were created using some data that was available to us from a group member's previous projects. Using Excel, the names and addresses were randomly scrambled to create a new dataset that could be imported into the database.

To create the bulk of the other data in the database which contained foreign key constraints, we took a couple of approaches as well. The first approach for generating this data was to use Excel to generate random numbers within a range contained in the fields that were foreign key constraints. We were able to do this because we created our entities with ID fields that were in numerical sequence. So, for example, if we were generating data for the `RESERVATION` table, we determined the range of the `CUSTOMER_ID` field and created a formula that would generate random numbers between the smallest value and the largest value in the `CUSTOMER_ID` field. This process was repeated for other fields where this process would work, such as `HOTEL_PROPERTY` and `ROOM_TYPE_ID`. The second approach in our extract, transform, and load (ETL) process to fill the other tables was to create a temporary table without any constraints that could house related data from other tables to be inserted into tables with constraints. This temporary table was used to combine data from the `CUSTOMER`, `HOTEL_PROPERTY`, `RESERVATION`, and `VISIT` tables in various ways in order to fill the `VISIT_ROOM_CUSTOMER` and `PAYMENT` tables with data that contained the correct foreign keys.

Using the above methods, we were able to create a significant amount of data for this project which is briefly summarized below:

Table Name	Number of Records
CUSTOMER	23,025
HOTEL_PROPERTY	105
PAYMENT	9,709
RESERVATION	9936
ROOM	10,500
VISIT	9,709

Query Writing

Here are a few of the queries we made for use with this database.

A general query to return details of customers with their stay information and hotel property information.

```
SELECT
    customer.customer_id,
    customer_fname,
    customer_address,
    customer_city,
    room_price,
    stay_length,
    hotel_property.hotel_property_id,
    hotel_property_name
FROM
    customer
    INNER JOIN payment
        ON ( customer.customer_id = payment.customer_id )
    INNER JOIN reservation
        ON ( payment.customer_id = reservation.customer_id )
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id =
hotel_property.hotel_property_id )
```

This query finds all customers who paid using an American Express credit card.

```
SELECT
    payment.customer_id,
    visit_id,
    payment_amount
FROM
    payment
        INNER JOIN payment_type
            ON ( payment.payment_type_id =
payment_type.payment_type_id )
WHERE
    payment_type_name LIKE 'AMEX'
```

This query returns any states that do not have hotel properties. Depending on guest feedback, it may prove to be fruitful to build properties in these states.

```
SELECT
    state.state_name
```

```
FROM
    hotel_property
    RIGHT OUTER JOIN state
        ON hotel_property.hotel_property_state_id = state.state_id
WHERE
    hotel_property_id IS NULL;
```

This query will return the most popular hotel in the month of May this year.

```
SELECT
    hotel_property.hotel_property_name,
    COUNT(reservation_id)
FROM
    reservation
    JOIN hotel_property
        ON reservation.hotel_property_id =
hotel_property.hotel_property_id
WHERE
    reservation_check_in
        BETWEEN TO_DATE('2019-05-01', 'yyyy-mm-dd')
            AND TO_DATE('2019-05-30', 'yyyy-mm-dd')
GROUP BY
    hotel_property.hotel_property_name
ORDER BY
    COUNT(reservation_id) DESC;
```

This query fetches the total number of customers that stayed in a given city at a given hotel.

```
SELECT DISTINCT
    customer_city,
    hotel_property.hotel_property_name,
    COUNT(*) AS "TOTAL GUESTS"
FROM
    customer
    INNER JOIN payment
        ON ( customer.customer_id = payment.customer_id )
    INNER JOIN reservation
        ON ( payment.customer_id = reservation.customer_id )
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id =
hotel_property.hotel_property_id )
GROUP BY
    customer_city,
    hotel_property.hotel_property_name
ORDER BY
```

```
    COUNT(*) DESC
```

This query fetches the preferred rooms in hotels. For this query, preference means that a hotel room was visited more than 20 times.

```
SELECT DISTINCT
    reservation.hotel_property_id,
    reservation.room_type_id,
    hotel_property_name,
    COUNT(*) AS count_rooms
FROM
    reservation
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id =
hotel_property.hotel_property_id )
GROUP BY
    reservation.hotel_property_id,
    reservation.room_type_id,
    hotel_property_name
HAVING
    COUNT(*) > 20
ORDER BY
    hotel_property_id ASC
```

This query finds the total amount a customer paid for each of their reservations based on how long they stayed and the rate of the room they reserved.

```
SELECT
    reservation.customer_id,
    reservation_id,
    ( reservation_check_out - reservation_check_in ) *
room_type.room_type_price AS "PAYMENT"
FROM
    reservation
    JOIN room_type
        ON reservation.room_type_id = room_type.room_type_id
    JOIN customer
        ON reservation.customer_id = customer.customer_id
ORDER BY
    reservation_id;
```

Variation of the previous query to find the total amount a customer has paid.

```
SELECT
    reservation.customer_id,
```

```
    SUM((reservation_check_out - reservation_check_in) *
room_type.room_type_price) AS "TOTAL PAYMENT"
FROM
    reservation
    JOIN room_type
        ON reservation.room_type_id = room_type.room_type_id
    JOIN customer
        ON reservation.customer_id = customer.customer_id
GROUP BY
    reservation.customer_id
ORDER BY
    reservation.customer_id;
```

Performance Tuning

We plan to use the SQL Tuning Advisor to improve the queries from the previous section.

The SQL Tuning Advisor had some index recommendations for the following query:

```
SELECT
    hotel_property.hotel_property_name,
    COUNT(reservation_id)
FROM
    reservation
    JOIN hotel_property
        ON reservation.hotel_property_id =
hotel_property.hotel_property_id
WHERE
    reservation_check_in
        BETWEEN TO_DATE('2019-05-01', 'yyyy-mm-dd')
            AND TO_DATE('2019-05-30', 'yyyy-mm-dd')
GROUP BY
    hotel_property.hotel_property_name
ORDER BY
    COUNT(reservation_id) DESC;
```

It suggested creating an index on the two columns involved in the query, `RESERVATION_CHECK_IN` and `HOTEL_PROPERTY_ID`.

Here is the index the advisor suggested:

```
CREATE INDEX db3white.idx$$_b1190001 ON
    db3white.reservation (
        "RESERVATION_CHECK_IN",
        "HOTEL_PROPERTY_ID"
    )
        TABLESPACE students;
```

Before creating the index, `Consistent gets` was `64`, and it dropped to `5` after creating the index. Performance was improved by creating the index.

Here is another approach to improving the performance of that same query.

```
SELECT
    hotel_property.hotel_property_name,
    COUNT(reservation_id)
FROM
    reservation
    JOIN hotel_property
```

```
        ON reservation.hotel_property_id =
hotel_property.hotel_property_id
WHERE
    reservation_check_in
        BETWEEN TO_DATE('2019-05-01', 'yyyy-mm-dd')
            AND TO_DATE('2019-05-30', 'yyyy-mm-dd')
GROUP BY
    hotel_property.hotel_property_name
ORDER BY
    COUNT(reservation_id) DESC;
```

We'll make an index on the `RESERVATION_CHECK_IN` and `RESERVATION_CHECK_OUT` columns.

```
CREATE INDEX db3white.date_query ON
    db3white.reservation (
        reservation_check_in,
        reservation_check_out
    )
        TABLESPACE students;
```

Before the index was created:

```
CONSISTENT GETS 764
DB block gets   0
```

After the index was created:

```
CONSISTENT GETS 68
DB block gets   0
```

Performance for this query was improved by using the `RESERVATION_CHECK_IN` and `RESERVATION_CHECK_OUT` dates as the index. Below are the screenshots.

```
SELECT
COUNT(reservation_id),
hotel_property_id
FROM
reservation
WHERE
reservation_check_in
BETWEEN TO_DATE('2019-05-01', 'yyyy-mm-dd')
AND TO_DATE('2019-05-30', 'yyyy-mm-dd')
GROUP BY
hotel_property_id
ORDER BY
COUNT(reservation_id) DESC;
```

Query Result × | Autotrace ×

SQL HotSpot | 20.706 seconds

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
SORT		ORDER BY

V$STATNAME Name	V$MYSTAT Value
cell physical IO interconnect bytes	319488
cluster key scan block gets	24
cluster key scans	18
consistent gets	784
consistent gets examination	311
consistent gets examination (fastpath)	310
consistent gets from cache	784
consistent gets pin	473

```
SELECT
COUNT(reservation_id),
hotel_property_id
FROM
reservation
WHERE
reservation_check_in
BETWEEN TO_DATE('2019-05-01', 'yyyy-mm-dd')
AND TO_DATE('2019-05-30', 'yyyy-mm-dd')
GROUP BY
hotel_property_id
ORDER BY
COUNT(reservation_id) DESC;
select * from reservation
CREATE INDEX db3white.date_query ON
    db3white.reservation ( Reservation_check_in,reservation_check_out)
        tablespace students;
```

Script Output × | Query Result × | Autotrace ×

SQL HotSpot | 19.224 seconds

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		
SORT		ORDER BY

V$STATNAME Name	V$MYSTAT Value
calls to get snapshot scn: kcmgss	10
calls to kcmgcs	11
consistent gets	68
consistent gets from cache	68
consistent gets pin	68

This query fetches the total number of customers in all hotels at city “RUSKIN”. Let’s try to improve its performance.

```
SELECT DISTINCT
    customer_city,
    hotel_property.hotel_property_name,
    COUNT(*) AS "TOTAL GUESTS"
FROM
    customer
    INNER JOIN payment
        ON ( customer.customer_id = payment.customer_id )
    INNER JOIN reservation
        ON ( payment.customer_id = reservation.customer_id )
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id =
hotel_property.hotel_property_id )
        where customer_city like 'RUSKIN'
GROUP BY
    customer_city,
    hotel_property.hotel_property_name
ORDER BY
    COUNT(*) DESC
```

We’ll make an index on `CUSTOMER_CITY`:

```
CREATE INDEX db3white.customer_city_btree ON
    db3white.customer (
        customer_city
    )
        TABLESPACE students;
```

Before the index was created:

```
CONSISTENT GETS 368
DB block gets   0
```

After the index was created:

```
CONSISTENT GETS 184
DB block gets   0
```

Performance for this query was improved by using `CUSTOMER_CITY` as an index. Below are the screenshots.

```
SELECT DISTINCT
    customer_city,
    hotel_property.hotel_property_name,
    COUNT(*) AS "TOTAL GUESTS"
FROM
    customer
    INNER JOIN payment
        ON ( customer.customer_id = payment.customer_id )
    INNER JOIN reservation
        ON ( payment.customer_id = reservation.customer_id )
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id = hotel_property.hotel_property_id )
        where customer_city like 'RUSKIN'
GROUP BY
    customer_city,
    hotel_property.hotel_property_name
ORDER BY
    COUNT(*) DESC
```

Script Output | Query Result | Autotrace

SQL HotSpot | 19.135 seconds

OPERATION	OBJECT_NAME	OPTIONS
SELECT STATEMENT		

V$STATNAME Name	V$MYSTAT Value
calls to get snapshot scn: kcmgss	13
calls to kcmgcs	22
consistent gets	368
consistent gets from cache	368
consistent gets pin	368

```
CREATE INDEX db3white.customer_city_btree ON
    db3white.customer ( customer_city)
        tablespace students;

        SELECT DISTINCT
    customer_city,
    hotel_property.hotel_property_name,
    COUNT(*) AS "TOTAL GUESTS"
FROM
    customer
    INNER JOIN payment
        ON ( customer.customer_id = payment.customer_id )
    INNER JOIN reservation
        ON ( payment.customer_id = reservation.customer_id )
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id = hotel_property.hotel_property_id )
        where customer_city like 'RUSKIN'
GROUP BY
    customer_city,
    hotel_property.hotel_property_name
ORDER BY
    COUNT(*) DESC
```

Script Output | Query Result | Autotrace

SQL HotSpot | 19.084 seconds

OPERATION	OBJECT_NAME	OPTIONS

V$STATNAME Name	V$MYSTAT Value
calls to kcmgcs	18
consistent gets	183

This query finds all customers who paid using an American Express credit card.
Let's try to improve this query's performance.

```
SELECT
    payment.customer_id,
    visit_id,
    payment_amount
FROM
    payment
        INNER JOIN payment_type
            ON ( payment.payment_type_id =
payment_type.payment_type_id )
WHERE
    payment_type_name LIKE 'AMEX'
```

We'll make an index on `PAYMENT_TYPE_NAME`:

```
CREATE INDEX db3white.payment_type_btree ON
    db3white.payment_type (
        payment_type_name
    )
        TABLESPACE students;
```

Before the index was created:

```
CONSISTENT GETS 15
DB block gets   0
```

After the index was created:

```
CONSISTENT GETS 10
DB block gets   0
```

Performance for this query was improved by using `PAYMENT_TYPE_NAME` as an index. Below are the screenshots.

```
select payment.customer_id,visit_id,payment_amount
from payment inner join payment_type on(payment.payment_type_id = payment_type.payment_type_id)
where PAYMENT_TYPE_NAME LIKE 'AMEX'
```

Script Output × | Query Result × | Autotrace ×

SQL HotSpot | 18.315 seconds

OPERATION	OBJECT_NAME	OPTIONS	CAR
SELECT STATEMENT			
HASH JOIN			

V$STATNAME Name	V$MYSTAT Value
buffer is not pinned count	8
bytes received via SQL*Net from client	2383
bytes sent via SQL*Net to client	52697
calls to get snapshot scn: kcmgss	7
calls to kcmgcs	10
consistent gets	15
consistent gets from cache	15
consistent gets pin	15

After indexing :

```
CREATE INDEX db3white.payment_type_btree ON
    db3white.payment_type ( payment_type_name)
        tablespace students;

select payment.customer_id,visit_id,payment_amount
from payment inner join payment_type on(payment.payment_type_id = payment_type.payment_type_id)
where PAYMENT_TYPE_NAME LIKE 'AMEX'
```

Script Output × | Query Result × | Autotrace ×

SQL HotSpot | 18.923 seconds

OPERATION	OBJECT_NAME	OPTIONS	CARDINALIT
SELECT STATEMENT			
HASH JOIN			

V$STATNAME Name	V$MYSTAT Value
buffer is not pinned count	11
bytes received via SQL*Net from client	2492
bytes sent via SQL*Net to client	52985
calls to get snapshot scn: kcmgss	12
calls to kcmgcs	9
consistent gets	10
consistent gets from cache	10
consistent gets pin	10

The two queries that calculated the total price of a customer's reservation and the total amount a customer has paid for all of their reservations both had recommendations from the SQL Tuning Advisor, but they could not be put into action, as the group account doesn't have the appropriate privileges to accept the recommended execution plan.

This is the query the SQL Tuning Advisor suggested be run to use the recommended execution plan:

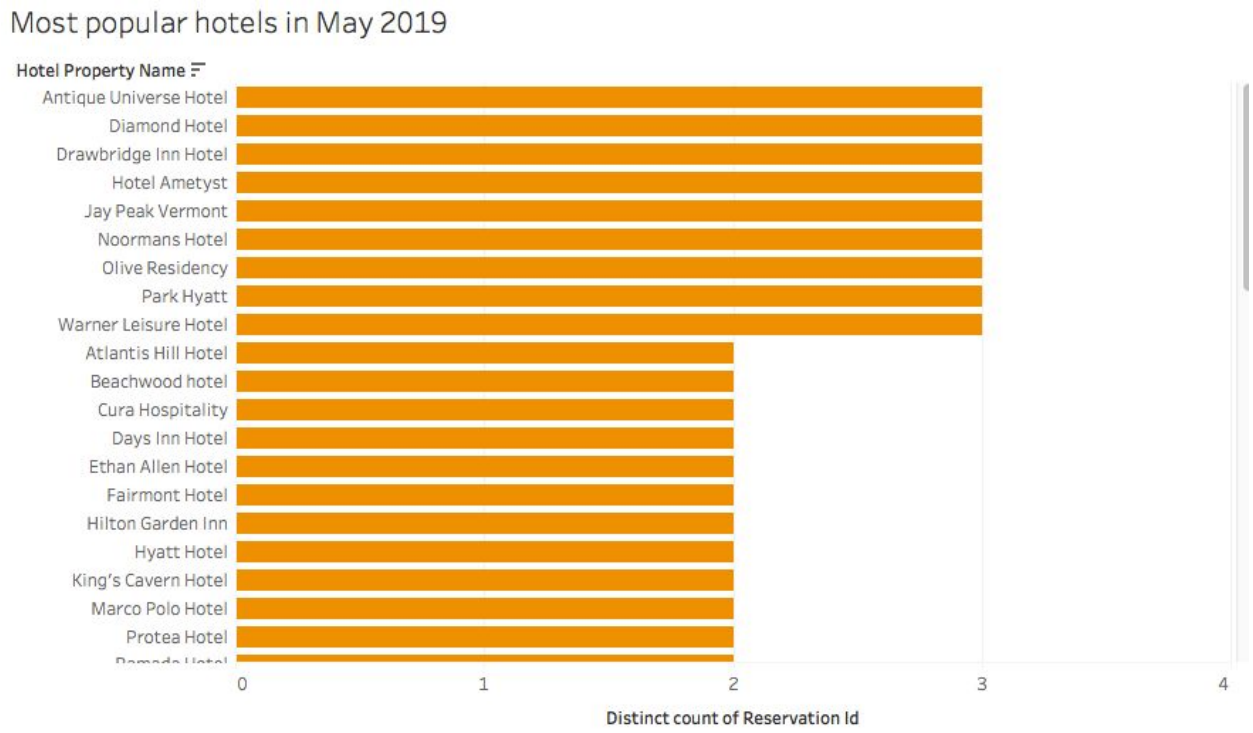
```
execute dbms_sqltune.accept_sql_profile(task_name => 'staName79216',
task_owner => 'DB3WHITE', replace => TRUE);
```

Other Topics

Data Visualization

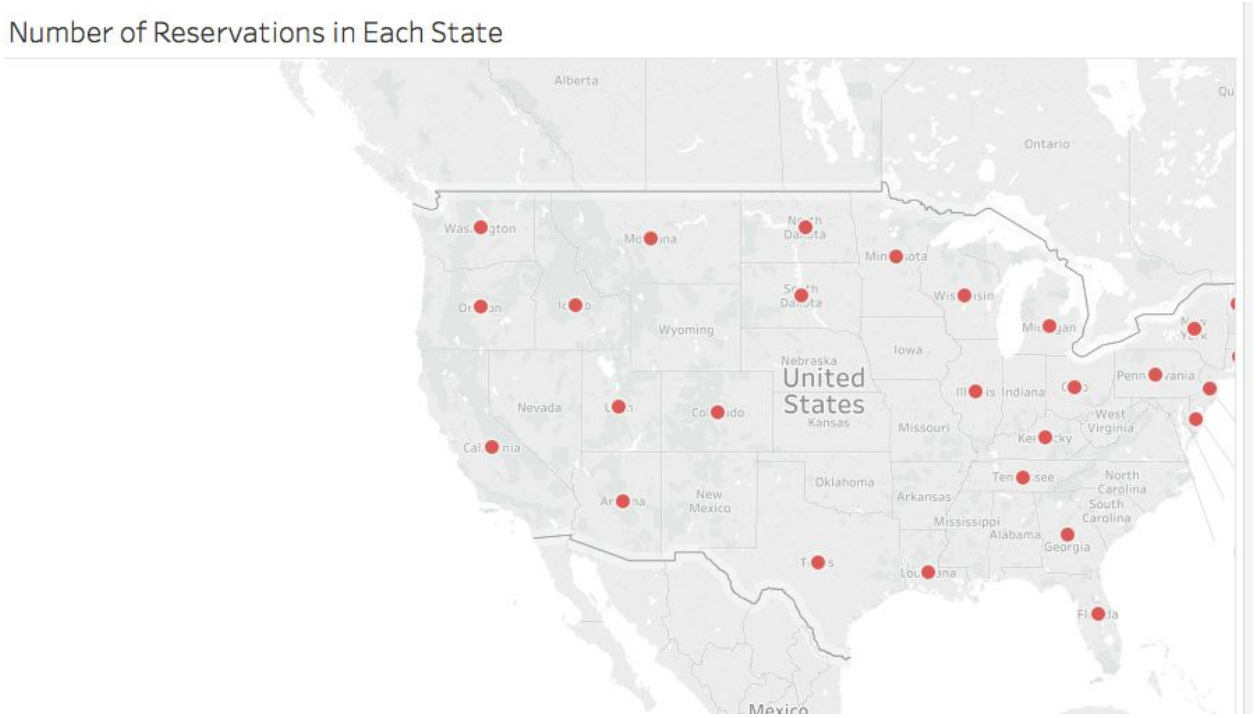
Query results can be shown using data visualization tools such as Tableau to better communicate trends to stakeholders. Here is the link for the interactive public Tableau page: https://public.tableau.com/profile/mary8174#!/vizhome/ISMFinalProjectSummer2019/Dashboard1.

Screenshots of each are shown below with their associated query.

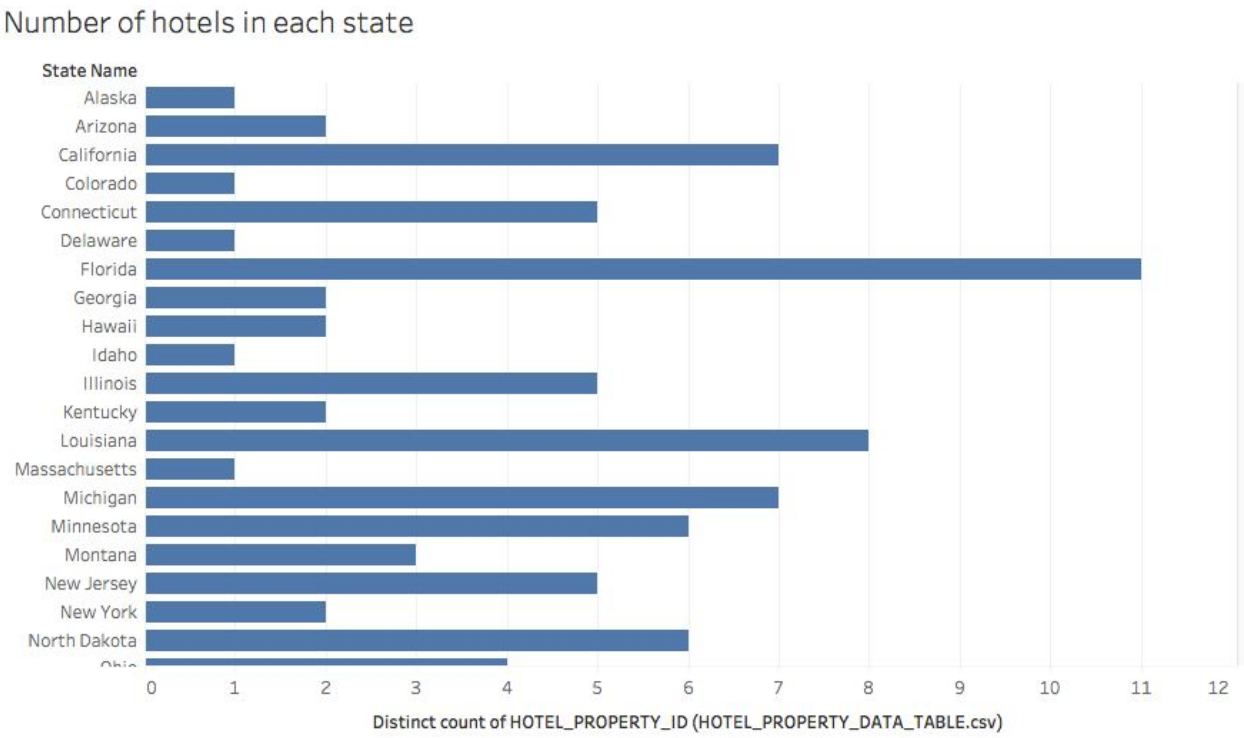



```
SELECT
    COUNT(reservation_id),
    hotel_property_name
FROM
    reservation
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id ) = (
hotel_property.hotel_property_id )
WHERE
    reservation_check_in
        BETWEEN TO_DATE('2019-05-01', 'yyyy-mm-dd') AND
TO_DATE('2019-05-30', 'yyyy-mm-dd')
GROUP BY
    hotel_property_name
```

```
ORDER BY
    COUNT(reservation_id) DESC;
```




```
SELECT
    COUNT(reservation_id),
    state_name
FROM
    reservation
    INNER JOIN hotel_property
        ON ( reservation.hotel_property_id ) = (
hotel_property.hotel_property_id )
    INNER JOIN state
        ON ( hotel_property.hotel_property_state_id = state.state_id
)
GROUP BY
    state_name;
```




```
SELECT
    COUNT(hotel_property.hotel_property_id),
    state_name
FROM
    hotel_property
    INNER JOIN state
        ON ( state_id = hotel_property.hotel_property_state_id )
GROUP BY
    state.state_name
ORDER BY
    state_name;
```

Data Mining

This data can be used to determine important business decision. As our guest tracking database can have clusters of users with their personal interest details which can be harnessed for marketing intelligence. One such attempt is clustering.

Assumption: We do not have any real actual data so results may not be accurate, but the process can be applied with the aid of additional personal data.

Preprocessing

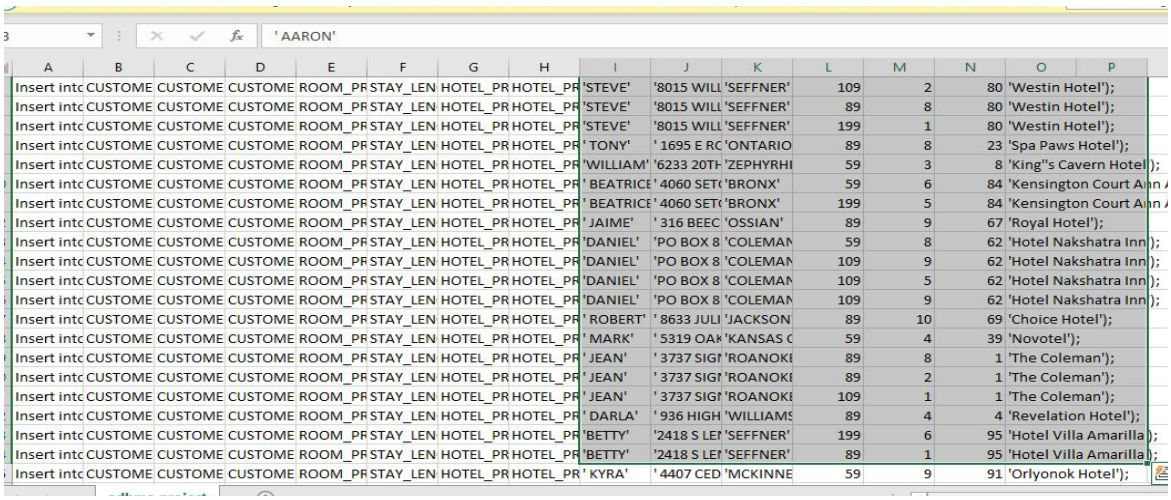
```
use table space student

select customer.customer_id, customer_fname, customer_address,customer_city,room_price,stay_length,hotel_property.hotel_property_id,hotel_proper
from
customer inner join payment on
(customer.customer_id = payment.customer_id)
inner join reservation on
(payment.customer_id= reservation.customer_id)
inner join hotel_property on
(reservation.hotel_property_id = hotel_property.hotel_property_id)
```

Autotrace | Script Output | Query Result

SQL | Fetched 50 rows in 0.335 seconds

	CUSTOMER_ID	CUSTOMER_FNAME	CUSTOMER_ADDRESS	CUSTOMER_CITY	ROOM_PRICE	STAY_LENGTH	HOTEL_PROPERTY_ID	HOTEL_PROPERTY_NAME
1	11231	AARON	408 WATSON DRIVE	NATCHITOCHES	199	3	33	Hilton Dusseldorf
2	11231	AARON	408 WATSON DRIVE	NATCHITOCHES	199	6	33	Hilton Dusseldorf
3	6262	STEVE	8015 WILLIAMS RD	SEFFNER	109	2	80	Westin Hotel
4	6262	STEVE	8015 WILLIAMS RD	SEFFNER	89	8	80	Westin Hotel
5	6262	STEVE	8015 WIL		199	1	80	Westin Hotel
6	14916	TONY	1695 E		89	8	23	Spa Paws Hotel
7	2954	WILLIAM	6233 20T	LLS	59	3	8	King's Cavern Hotel
8	16900	BEATRICE	4060 SE		59	6	84	Kensington Court Ann Arbor
9	16900	BEATRICE	4060 SE		199	5	84	Kensington Court Ann Arbor
10	9144	JAIME	316 BEE		89	9	67	Royal Hotel
11	4881	DANIEL	PO BOX 8		59	8	62	Hotel Nakshatra Inn

Save Grid as Report...
Publish to REST
Single Record View...
Count Rows...
Find/Highlight...
Export...



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'STEVE'	'8015 WILL	'SEFFNER'	109	2	80	'Westin Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'STEVE'	'8015 WILL	'SEFFNER'	89	8	80	'Westin Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'STEVE'	'8015 WILL	'SEFFNER'	199	1	80	'Westin Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' TONY'	' 1695 E RC	'ONTARIO	89	8	23	'Spa Paws Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'WILLIAM'	'6233 20TH	'ZEPHYRHI	59	3	8	'King''s Cavern Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' BEATRICE	' 4060 SET(	'BRONX'	59	6	84	'Kensington Court Ann A		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' BEATRICE	' 4060 SET(	'BRONX'	199	5	84	'Kensington Court Ann A		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' JAIME'	' 316 BEEC	'OSSIAN'	89	9	67	'Royal Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'DANIEL'	'PO BOX 8	'COLEMAN	59	8	62	'Hotel Nakshatra Inn');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'DANIEL'	'PO BOX 8	'COLEMAN	109	9	62	'Hotel Nakshatra Inn');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'DANIEL'	'PO BOX 8	'COLEMAN	109	5	62	'Hotel Nakshatra Inn');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'DANIEL'	'PO BOX 8	'COLEMAN	109	9	62	'Hotel Nakshatra Inn');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' ROBERT'	' 8633 JULI	'JACKSON	89	10	69	'Choice Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' MARK'	' 5319 OAK	'KANSAS C	59	4	39	'Novotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' JEAN'	' 3737 SIGI	'ROANOKE	89	8	1	'The Coleman');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' JEAN'	' 3737 SIGI	'ROANOKE	89	2	1	'The Coleman');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' JEAN'	' 3737 SIGI	'ROANOKE	109	1	1	'The Coleman');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' DARLA'	' 936 HIGH	'WILLIAMS	89	4	4	'Revelation Hotel');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'BETTY'	'2418 S LEI	'SEFFNER'	199	6	95	'Hotel Villa Amarilla');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	'BETTY'	'2418 S LEI	'SEFFNER'	89	1	95	'Hotel Villa Amarilla');		
Insert into	CUSTOME	CUSTOME	CUSTOME	ROOM_PR	STAY_LEN	HOTEL_PR	HOTEL_PR	' KYRA'	' 4407 CED	'MCKINNE	59	9	91	'Orlyonok Hotel');		

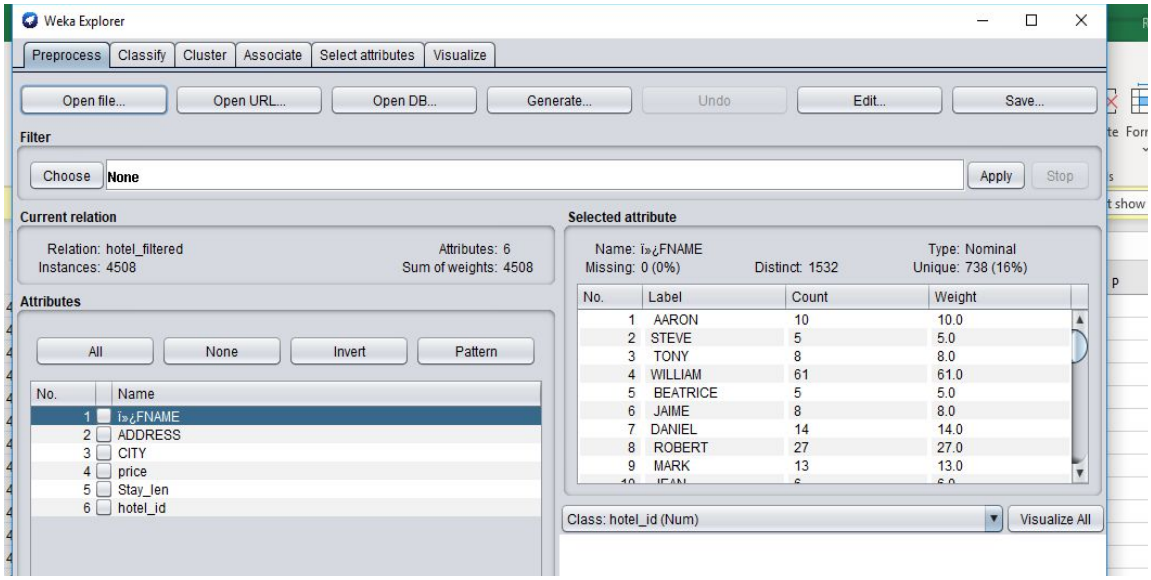
Took the sample of the data, roughly 5000 rows in other Excel sheet.

I4422

	A	B	C	D	E	F	G	H
1	FNAME	ADDRESS	CITY	price	Stay_len	hotel_id		
2	' AARON'	' 408 WAT	'NATCHITC	199	3	33		
3	' AARON'	' 408 WAT	'NATCHITC	199	6	33		
4	'STEVE'	'8015 WILL	'SEFFNER'	109	2	80		
5	'STEVE'	'8015 WILL	'SEFFNER'	89	8	80		
6	'STEVE'	'8015 WILL	'SEFFNER'	199	1	80		
7	' TONY'	' 1695 E RC	'ONTARIO	89	8	23		
8	'WILLIAM'	'6233 20TH	'ZEPHYRHI	59	3	8		
9	' BEATRICE	' 4060 SET	'BRONX'	59	6	84		
10	' BEATRICE	' 4060 SET	'BRONX'	199	5	84		
11	' JAIME'	' 316 BEEC	'OSSIAN'	89	9	67		
12	'DANIEL'	'PO BOX 8	'COLEMAN	59	8	62		
13	'DANIEL'	'PO BOX 8	'COLEMAN	109	9	62		
14	'DANIEL'	'PO BOX 8	'COLEMAN	109	5	62		
15	'DANIEL'	'PO BOX 8	'COLEMAN	109	9	62		
16	' ROBERT'	' 8633 JULI	'JACKSON	89	10	69		
17	' MARK'	' 5319 OAK	'KANSAS C	59	4	39		
18	' JEAN'	' 3737 SIGI	'ROANOKE	89	8	1		
19	' JEAN'	' 3737 SIGI	'ROANOKE	89	2	1		
20	' JEAN'	' 3737 SIGI	'ROANOKE	109	1	1		
21	' DARLA'	' 936 HIGH	'WILLIAMS	89	4	4		

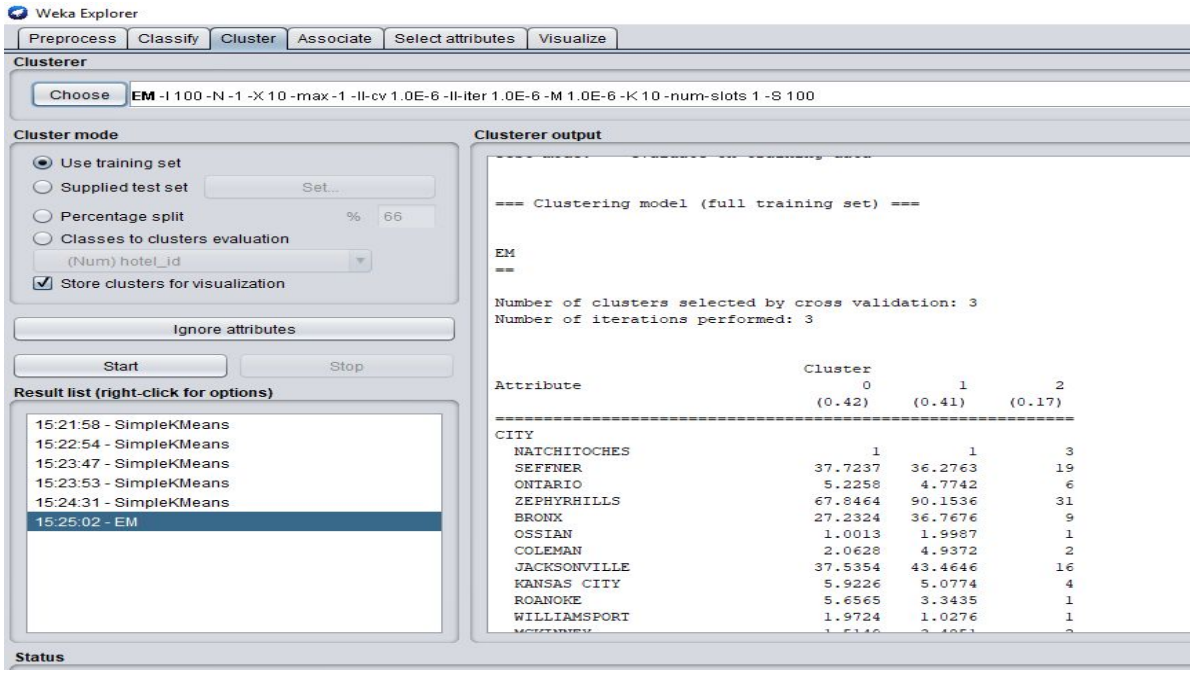
hotel_filtered

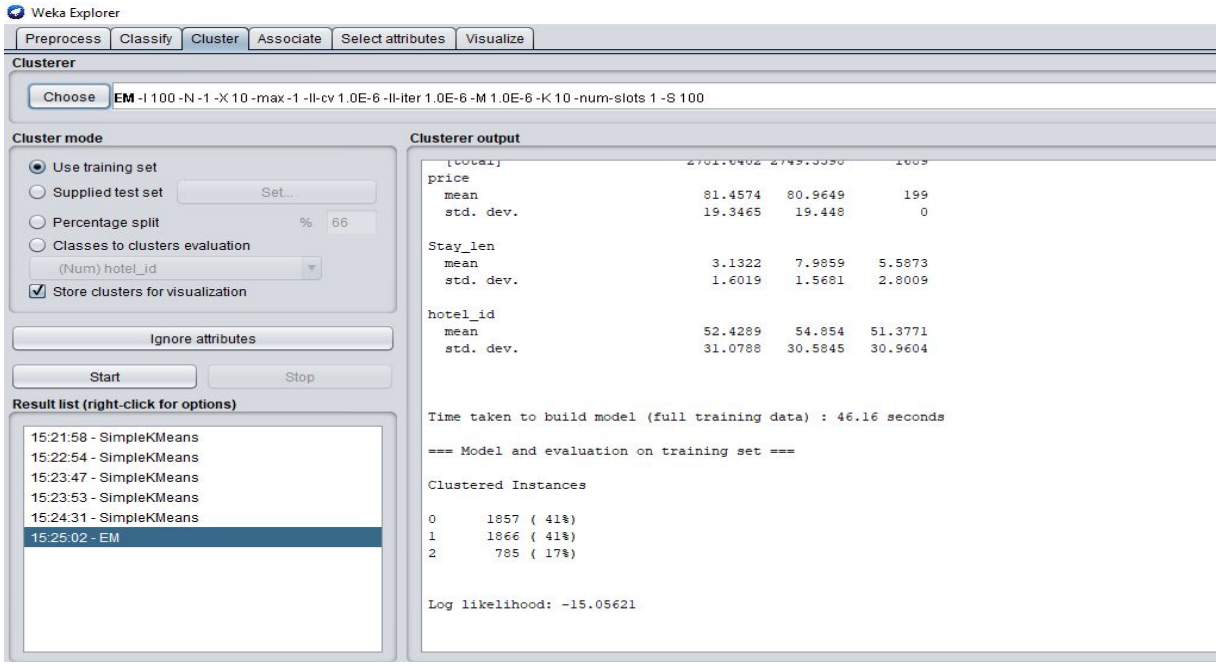
Inserted the file into Weka for Data Mining.



Removed the unnecessary columns, like address

Running Clustering





Basically, we have three clusters based on user with similar preferences along with the city they belong to and what hotel they choose.

This can be used to strategize about customer loyalty retention.