MONASH University

FIT 3003

Business Intelligence and Data Warehousing

Major Assignment - Assignment 2 - S2 2024 *by Mohib Ali Khan 33370311*

Data Exploration and Data Cleaning

To start with our exploration these are the SQL queries we executed:

```
-- Data Exploration: Display all data from each of the 11 tables in the MStay database
-- This step is to get an initial look at the data stored in each table.

SELECT * FROM MSTAY.booking;          -- Explore the booking table, which holds booking records
SELECT * FROM MSTAY.amenity;          -- Explore the amenity table, which stores amenities information
SELECT * FROM MSTAY.channel;          -- Explore the channel table, which stores information about booking channels
SELECT * FROM MSTAY.guest;            -- Explore the guest table, which holds guest records
SELECT * FROM MSTAY.host;             -- Explore the host table, which holds host records
SELECT * FROM MSTAY.host_verification;  -- Explore the host_verification table, which holds host verification details
SELECT * FROM MSTAY.listing;          -- Explore the listing table, which holds property listing records
SELECT * FROM MSTAY.listing_type;       -- Explore the listing_type table, which categorises the type of property listings
SELECT * FROM MSTAY.property;         -- Explore the property table, which holds specific property details
SELECT * FROM MSTAY.property_amenity;   -- Explore the property_amenity table, which links properties with their amenities
SELECT * FROM MSTAY.review;           -- Explore the review table, which holds reviews for listings

-- Data Summary: Count the number of records in each table in the MStay database
-- This provides an overview of the data volume in each table, which helps understand the size and scope of the data.

SELECT COUNT(*) AS total_rows FROM MSTAY.booking;       -- Booking table: 5002 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.amenity;       -- Amenity table: 449 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.channel;       -- Channel table: 20 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.guest;         -- Guest table: 9372 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.host;          -- Host table: 3883 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.host_verification;  -- Host Verification table: 21749 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.listing;       -- Listing table: 4936 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.listing_type;   -- Listing Type table: 4 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.property;      -- Property table: 5001 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.property_amenity;  -- Property Amenity table: 47027 Records
SELECT COUNT(*) AS total_rows FROM MSTAY.review;        -- Review table: 4870 Records

-- Notes:
-- - The 'Booking' table holds 5002 records, indicating that it has a large volume of transaction data.
-- - The 'Host' and 'Guest' tables show substantial data, with 9372 guest records and 3883 host records,
--   which will be useful for analysing guest-host relationships.
-- - The 'Listing' table contains the largest volume of data, with 21749 records, reflecting the various property listings.
-- - The 'Review' table has 4870 records, making it critical for sentiment analysis or feedback evaluation.
-- - Tables like 'Channel', 'Listing_Type' have smaller record counts, showing more static or categorical data.
```

Strategy 1: Duplication Issues

To dive deeper into the exploration of data the next aim is to discover potential **Duplication Issues**
This error arises when a record is inserted with a primary key value that already exists in the table.
To prevent duplicate key errors, it's essential to ensure that primary key values remain unique across all records.

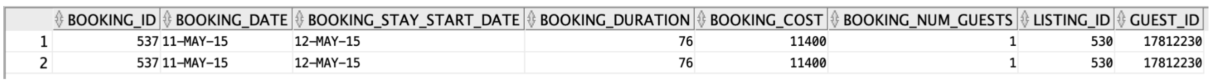
1.1

Exploration/Identifying Issue SQL Code:

```
–issue
select BOOKING_ID , count(*)
from Mstay.booking
group by BOOKING_ID
having count(*) > 1;

select * from
MStay.booking
where BOOKING_ID = 537;
```

Issue:

To address duplication issues, the first step is to identify records with duplicate primary key values using a GROUP BY and HAVING clause.
Upon which it was revealed that in the booking table the `booking_id` of **537** had a duplicate record refer to the screenshot attached below.



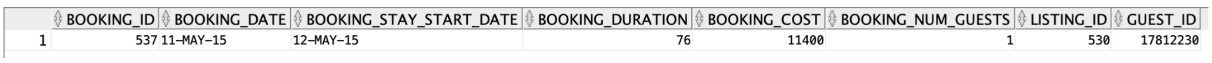
	BOOKING_ID	BOOKING_DATE	BOOKING_STAY_START_DATE	BOOKING_DURATION	BOOKING_COST	BOOKING_NUM_GUESTS	LISTING_ID	GUEST_ID
1	537	11-MAY-15	12-MAY-15	76	11400	1	530	17812230
2	537	11-MAY-15	12-MAY-15	76	11400	1	530	17812230

Solution/Data Cleaning SQL Code:

```
–Solution
drop table booking;
create table booking as
select distinct * from MStay.booking;

select booking_id, count(*)
from booking
group by booking_id
having count(*) > 1;

select * from booking
where BOOKING_ID = 537;
```

Solution:

After identifying the duplicates, a new table is created using the SELECT DISTINCT method to ensure only unique records are retained. This process eliminates any duplicates and preserves data integrity in the system (refer to the screenshot attached below) after we run the command using the GROUPBY and HAVING clause.



	BOOKING_ID	BOOKING_DATE	BOOKING_STAY_START_DATE	BOOKING_DURATION	BOOKING_COST	BOOKING_NUM_GUESTS	LISTING_ID	GUEST_ID
1	537	11-MAY-15	12-MAY-15	76	11400	1	530	17812230

1.2

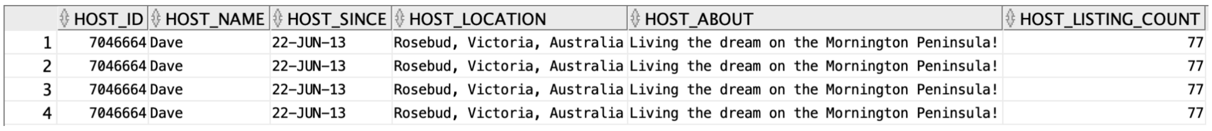
Continuing with same approach of using the GROUP BY and HAVING clause to identify duplicates in the remaining records it was revealed that the host table had the similar duplication issue

Exploration/Identifying Issue SQL Code:

```
–issue
select HOST_ID , count(*)
from Mstay.host
group by HOST_ID
having count(*) > 1;

select * from
MStay.host
where HOST_ID = 7046664;
```

Issue:

where `host_id` of 7046664 had duplicate records of a total 4 count refer to the screenshot attached below.



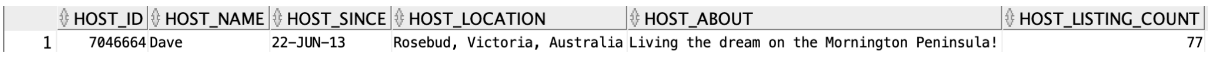
	HOST_ID	HOST_NAME	HOST_SINCE	HOST_LOCATION	HOST_ABOUT	HOST_LISTING_COUNT
1	7046664	Dave	22-JUN-13	Rosebud, Victoria, Australia	Living the dream on the Mornington Peninsula!	77
2	7046664	Dave	22-JUN-13	Rosebud, Victoria, Australia	Living the dream on the Mornington Peninsula!	77
3	7046664	Dave	22-JUN-13	Rosebud, Victoria, Australia	Living the dream on the Mornington Peninsula!	77
4	7046664	Dave	22-JUN-13	Rosebud, Victoria, Australia	Living the dream on the Mornington Peninsula!	77

Solution/Data Cleaning SQL Code:

```
–Solution
DROP TABLE host;
CREATE TABLE host AS
SELECT DISTINCT *
FROM MStay.host;

select host_id, count(*)
from host
group by HOST_ID
having count(*) > 1;

select * from host
where HOST_ID = 7046664;
```

Solution:

After identifying the duplicates, a new table is created using the SELECT DISTINCT method to ensure only unique records are retained.
This process eliminates any duplicates and preserves data integrity in the system. To check the cleaned data refer to the screenshot below.



	HOST_ID	HOST_NAME	HOST_SINCE	HOST_LOCATION	HOST_ABOUT	HOST_LISTING_COUNT
1	7046664	Dave	22-JUN-13	Rosebud, Victoria, Australia	Living the dream on the Mornington Peninsula!	77

Upon further exploration, it was determined that all tables, with the exception of the `booking` and `host` tables, did not contain any duplicate primary key records and this was explored through the same method used to check `booking` and `host` tables.

Strategy 2: Constraint Violations or Relationship Issues

Constraint Violations or Relationship Issues: Violations of constraints, such as primary key, foreign key, unique, or check constraints, can occur when the inserted or updated data does not conform to the established rules. To prevent these errors, it is crucial to validate the data against the defined constraints before performing any operations.

2.1

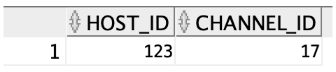
The goal of this step is to identify **referential integrity issues** by ensuring that foreign key values in one table correspond to existing primary key values in the related table. This type of error occurs when a foreign key value exists in a referencing table (like `host_verification`) but does not have a corresponding primary key in the referenced table (like `host` or `channel`).

Exploration/Identifying Issue SQL Code:

```
SELECT *
FROM Mstay.host_verification
WHERE host_id NOT IN (
    SELECT host_id
    FROM host
);
-- the host_id 123 does not exist in the host table but exists in host_verification table
select * from host
where host_id = 123;

SELECT *
FROM Mstay.host_verification
WHERE channel_id NOT IN (
    SELECT channel_id
    FROM Mstay.channel
);
-- the channel_id 17 does not exist in the channel table but exists in host_verification table
select * from Mstay.channel
where channel_id = 17;
```

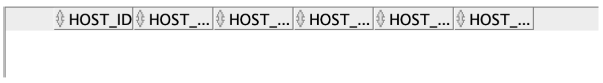
Issue:

In the `host_verification` table, it was discovered that some `host_id` and `channel_id` values did not have corresponding records in the `host` and `channel` tables. This violates referential integrity and suggests that there are foreign key values in `host_verification` that should not exist. Refer to the screenshot below which shows the data before cleaning when the first block of the above mentioned SQL code was executed.



	HOST_ID	CHANNEL_ID
1	123	17

The second screenshot shows that when a check was performed to explore if `host_id` = 123 exists in the `host` table and it was confirmed that it does not.



HOST_ID	HOST_...	HOST_...	HOST_...	HOST_...	HOST_...

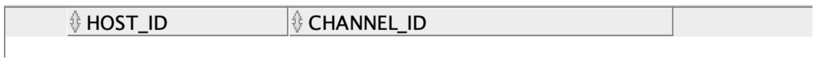
- **host_id = 123** exists in `host_verification` but does not exist in the `host` table.
- **channel_id = 17** exists in `host_verification` but does not exist in the `channel` table.

Solution/Data Cleaning SQL Code:

```
-- Solution
drop table host_verification;
create table host_verification as
select * from Mstay.host_verification;

DELETE
FROM host_verification
WHERE host_id NOT IN
        (SELECT host_id
        FROM host);
-- Rechecking if the data has been cleaned
SELECT *
FROM host_verification
WHERE host_id NOT IN
        (SELECT host_id
        FROM host);
```

Solution:

To resolve this issue, the problematic rows in `host_verification` where foreign keys do not have corresponding primary keys in the referenced tables (`host` and `channel`) must be removed to maintain data consistency. The screenshot below shows that when we remove the data from the `host_verification` table we recheck to explore if the data has been cleaned in the new table.



HOST_ID	CHANNEL_ID

2.2

The goal of this step is to again address the **referential integrity violations** by ensuring that all `booking_id` values in the `review` table correspond to valid `booking_id` values in the `booking` table. This type of error occurs when a `booking_id` is referenced in the `review` table but does not exist in the `booking` table. Ensuring that foreign keys in the `review` table refer to valid records in the `booking` table is critical for maintaining database integrity.

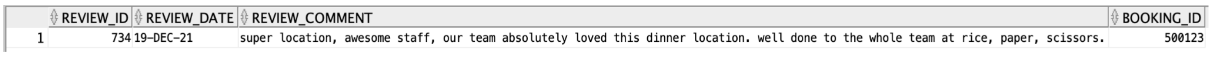
Exploration/Identifying Issue SQL Code:

```
–issue
SELECT *
FROM Mstay.review
WHERE booking_id NOT IN (
   SELECT booking_id
   FROM booking
);
-- the booking_id 500123 does not exist in booking table but is present in review table

select * from
MStay.booking
where BOOKING_ID = 500123;
```

Issue:

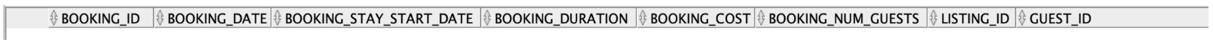
Upon investigation, it was discovered that the `review` table contains references to `booking_id` values that do not exist in the `booking` table. The `booking_id = 500123` exists in the `review` table but not in the `booking` table. This suggests the presence of orphaned foreign key references in the `review` table, which violates the referential integrity between the two tables.

The screenshot below shows the existence of `booking_id = 500123` in the `review` table.



	REVIEW_ID	REVIEW_DATE	REVIEW_COMMENT	BOOKING_ID
1	734	19-DEC-21	super location, awesome staff, our team absolutely loved this dinner location. well done to the whole team at rice, paper, scissors.	500123

but the same `booking_id = 500123` when searched in the `booking` table we get the result in the screenshot attached below.



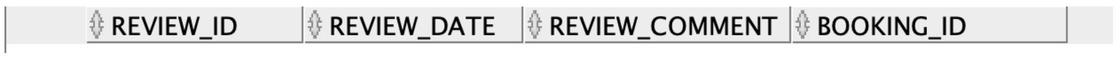
BOOKING_ID	BOOKING_DATE	BOOKING_STAY_START_DATE	BOOKING_DURATION	BOOKING_COST	BOOKING_NUM_GUESTS	LISTING_ID	GUEST_ID

Solution/Data Cleaning SQL Code:

```
-- Solution

drop table review;
create table review as
select * from Mstay.review;

DELETE
FROM review
WHERE booking_id NOT IN
            (SELECT booking_id
            FROM booking);

-- Rechecking if the data has been cleaned
SELECT *
FROM review
WHERE booking_id NOT IN
            (SELECT booking_id
            FROM booking);
```

Solution:

To resolve this issue, orphaned foreign key records (i.e., `booking_id` values in the `review` table that do not exist in the `booking` table) must be removed. This ensures that every review has a valid booking associated with it. The screenshot below shows that the `booking_id = 500123` is now removed from the `review` table.



REVIEW_ID	REVIEW_DATE	REVIEW_COMMENT	BOOKING_ID

2.3

Following the same strategy now we resolve **referential integrity violations** in the `listing` table by ensuring that foreign key values in the `listing` table correspond to valid primary key values in the `listing_type`, `host`, and `property` tables. Issues arise when `type_id`, `host_id`, or `prop_id` values in the `listing` table do not have corresponding records in the respective reference tables.

Exploration/Identifying Issue SQL Code:

```
SELECT *
FROM Mstay.listing
WHERE type_id NOT IN (
    SELECT type_id
    FROM Mstay.listing_type
);

SELECT *
FROM Mstay.listing
WHERE host_id NOT IN (
    SELECT host_id
    FROM host
);
-- the host_id 9999 does not exist in the host table but is present in listing table
SELECT *
FROM Mstay.listing
WHERE prop_id NOT IN (
    SELECT prop_id
    FROM Mstay.property
);
-- the prop_id 9999 does not exist in the property table but exists in listing table
```

Issue:

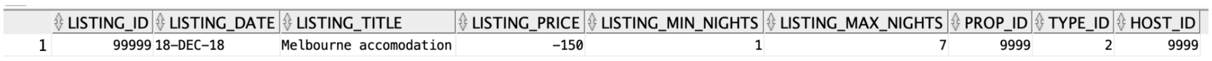
Upon running the above SQL code checks, it was discovered that the `listing` table contains invalid foreign key references:

- **host_id = 9999** exists in the `listing` table but not in the `host` table.
- **prop_id = 9999** exists in the `listing` table but not in the `property` table.

These discrepancies indicate broken foreign key relationships, leading to inconsistencies in the data.

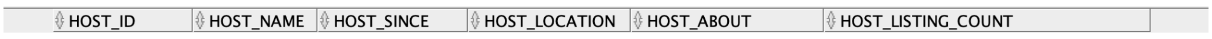
Refer to the screenshot below to observe the dirty data

but the same `host_id = 9999` when searched in the `host` table we get the result in the screenshot attached below.



	LISTING_ID	LISTING_DATE	LISTING_TITLE	LISTING_PRICE	LISTING_MIN_NIGHTS	LISTING_MAX_NIGHTS	PROP_ID	TYPE_ID	HOST_ID
1	99999	18-DEC-18	Melbourne accomodation	-150	1	7	9999	2	9999

but the same `prop_id = 9999` when searched in the `property` table we get the result in the screenshot attached below.



HOST_ID	HOST_NAME	HOST_SINCE	HOST_LOCATION	HOST_ABOUT	HOST_LISTING_COUNT

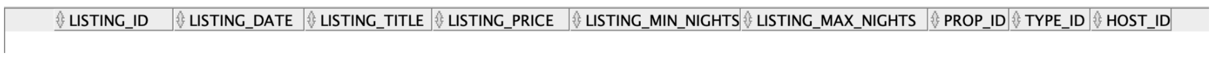
Solution/Data Cleaning SQL Code:

```
-- Solution
drop table listing;
create table listing as
select * from Mstay.listing;

DELETE
FROM listing
WHERE prop_id NOT IN
        (SELECT prop_id
        FROM Mstay.property);

-- Rechecking if the data has been cleaned
SELECT *
FROM listing
WHERE prop_id NOT IN
        (SELECT prop_id
        FROM Mstay.property);
```

Solution:

To resolve these referential integrity violations, the invalid foreign key entries (i.e., `type_id`, `host_id`, and `prop_id`) in the `listing` table must be cleaned up by removing the invalid records.

The screenshot below shows the result of rechecking block of code to check on the clean data



LISTING_ID	LISTING_DATE	LISTING_TITLE	LISTING_PRICE	LISTING_MIN_NIGHTS	LISTING_MAX_NIGHTS	PROP_ID	TYPE_ID	HOST_ID

Strategy 3: Null Values

In the data cleaning process, handling **NULL values** is crucial. NULL values represent missing data points, and identifying how to handle these values is essential for ensuring data accuracy, analysis reliability, and database integrity. The objective of this strategy is to locate NULL values in critical columns and either address or remove them depending on their relevance.

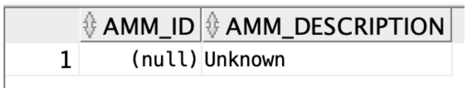
3.1

Exploration/Identifying Issue SQL Code:

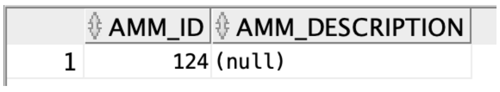
```
– Issue
-- Check for NULL in AMM_ID
SELECT *
FROM Mstay.Amenity
WHERE AMM_ID IS NULL;          ---null found

-- Check for NULL in AMM_DESCRIPTION
SELECT *
FROM Mstay.Amenity
WHERE AMM_DESCRIPTION IS NULL;
```

Issue:

NULL values were identified in both the AMM_ID and AMM_DESCRIPTION columns of the Amenity table. These fields are critical for uniquely identifying and describing each amenity. If left unresolved, NULL values can compromise the integrity of the data and the results of future analyses. The screenshots below shows the dirty data with NULL values



	AMM_ID	AMM_DESCRIPTION
1	(null)	Unknown



	AMM_ID	AMM_DESCRIPTION
1	124	(null)

Solution/Data Cleaning SQL Code:

```
-- Solution
drop table amenity
create table Amenity as
select * from Mstay.Amenity;

DELETE
FROM amenity
WHERE amm_id IS NULL;

DELETE
FROM amenity
WHERE amm_description IS NULL;

-- Re-Check for NULL in AMM_ID
SELECT *
FROM Amenity
WHERE AMM_ID IS NULL;

-- Re-Check for NULL in AMM_DESCRIPTION
SELECT *
FROM Amenity
WHERE AMM_DESCRIPTION IS NULL;
```

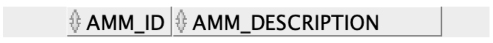
Solution:

The strategy to resolve this issue involved:

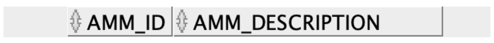
1. **Dropping and recreating the Amenity table**: This step is essential to work with a clean table structure.
2. The Amenity table contains only two key columns: AMM_ID, which uniquely identifies each amenity, and AMM_DESCRIPTION, which provides details about the amenity. The presence of NULL values in either of these columns compromises the purpose of the data. Therefore, the decision was made to delete rows with NULL values in both columns, for the following reasons:
    - **Missing AMM_ID**: If an amenity lacks an ID (AMM_ID), the corresponding description (AMM_DESCRIPTION) becomes irrelevant. Without a valid identifier, the record is incomplete and unusable, making it necessary to delete such rows.
    - **Missing AMM_DESCRIPTION**: If the description (AMM_DESCRIPTION) is missing, then the AMM_ID itself becomes less meaningful. The ID alone doesn't provide any insight about the amenity, and storing such rows wastes space without adding value. To ensure efficient data storage, rows with missing descriptions were also removed.

3. **Rechecking**: A final check was done to ensure no further NULL values were present in either of the columns after the cleanup.

The screenshots below is the cleaned data where there are no null values found in each column



AMM_ID	AMM_DESCRIPTION



AMM_ID	AMM_DESCRIPTION

3.2

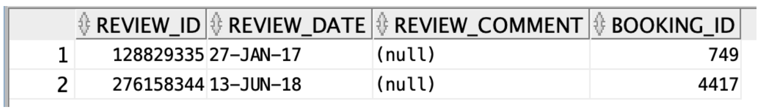
In further exploration of data to track null values in the Review table, the goal is to handle **NULL values** present in critical columns within the `Review` table.

Exploration/Identifying Issue SQL Code:

```
–Issue
– Check for NULL in REVIEW_ID
SELECT *
FROM Review
WHERE REVIEW_ID IS NULL;

-- Check for NULL in REVIEW_DATE
SELECT *
FROM Review
WHERE REVIEW_DATE IS NULL;

-- Check for NULL in REVIEW_COMMENT
SELECT *
FROM Review
WHERE REVIEW_COMMENT IS NULL;       ---  null found
```

Issue:

During the process of data cleaning, it was found that some rows in the `Review` table had `NULL` values in the `REVIEW_COMMENT` column. Missing comments can undermine the quality of feedback or analysis based on reviews, as comments usually provide meaningful insights as the point of the review table is primarily to store the reviews. The screenshot below shows the dirty data with NULL values.



	REVIEW_ID	REVIEW_DATE	REVIEW_COMMENT	BOOKING_ID
1	128829335	27-JAN-17	(null)	749
2	276158344	13-JUN-18	(null)	4417

Solution/ Data Cleaning SQL Code:

```
-- Solution
UPDATE Review
SET Review_Comment = 'No Comment'
WHERE Review_Comment IS NULL;

-- Re-Check for NULL in REVIEW_COMMENT
SELECT *
FROM Review
WHERE REVIEW_COMMENT IS NULL;
```

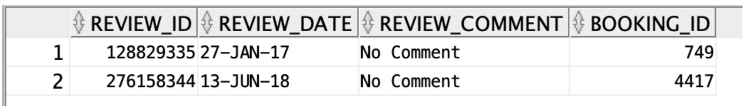
```
-- Check for NULL in BOOKING_ID
SELECT *
FROM Review
WHERE BOOKING_ID IS NULL;
```

Solution:

To address this issue, the following steps were taken:

1. **Check for NULL Values**:
   - The first step was to identify any `NULL` values in the critical columns like `REVIEW_COMMENT`.
2. **Replace NULL with Default Value**:
   - The `NULL` values in the `REVIEW_COMMENT` column were replaced with a default comment, `'No Comment'`, indicating that the review lacks additional textual feedback but ensuring no missing values remain in the dataset.
3. **Rechecking**:
   - After the update, a recheck was done to ensure that all `NULL` values in the `REVIEW_COMMENT` column were successfully replaced.

You can view the cleaned data below in the screenshot attached below



	REVIEW_ID	REVIEW_DATE	REVIEW_COMMENT	BOOKING_ID
1	128829335	27-JAN-17	No Comment	749
2	276158344	13-JUN-18	No Comment	4417

3.3

Exploration/Identifying Issue SQL Code:

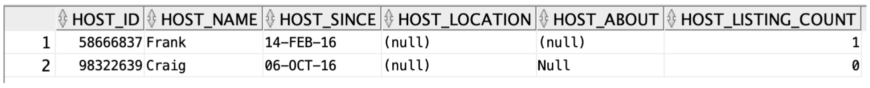
```
-- Check for NULL in HOST_ID
SELECT *
FROM Host
WHERE HOST_ID IS NULL;

-- Check for NULL in HOST_NAME
SELECT *
FROM Host
WHERE HOST_NAME IS NULL;

-- Check for NULL in HOST_SINCE
SELECT *
FROM Host
WHERE HOST_SINCE IS NULL;

-- Check for NULL in HOST_LOCATION
SELECT *
FROM Host
WHERE HOST_LOCATION IS NULL; ---     Null Found

-- Check for NULL in HOST_ABOUT
SELECT *
FROM Host
WHERE HOST_ABOUT IS NULL; ----     Null Found
```

Issue:

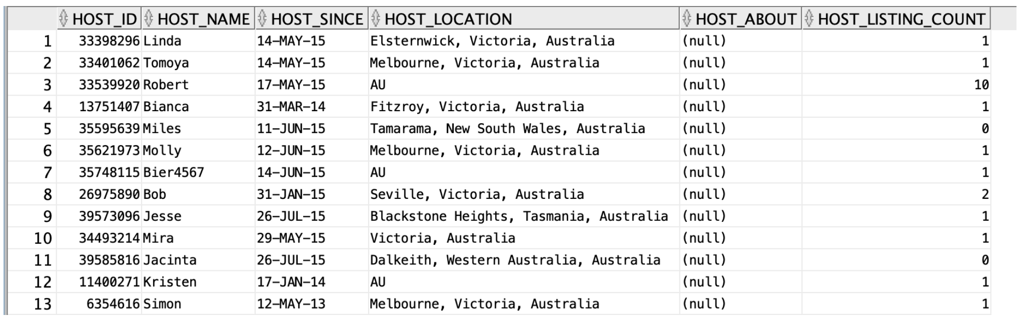
During the data exploration, it was observed that the `Host_Location` column contains `NULL` values in several rows. This is particularly problematic as the location of a host is a critical attribute in any booking or listing system. Missing location data can significantly reduce the effectiveness of analyses that rely on geographic information, thereby diminishing the overall usefulness of the data.

Additionally, while the `Host_About` column is not considered critical information, it was found that approximately 1,380 rows contained `NULL` values. To ensure data consistency, these `NULL` values were replaced with the default entry "No About." This approach ensures that all records in the column now contain a value, maintaining the integrity of the dataset without impacting the system's overall functionality.

The screenshots below shows the records will `NULL` values in `Host_Location` and `Host_About`



	HOST_ID	HOST_NAME	HOST_SINCE	HOST_LOCATION	HOST_ABOUT	HOST_LISTING_COUNT
1	58666837	Frank	14-FEB-16	(null)	(null)	1
2	98322639	Craig	06-OCT-16	(null)	Null	0



	HOST_ID	HOST_NAME	HOST_SINCE	HOST_LOCATION	HOST_ABOUT	HOST_LISTING_COUNT
1	33398296	Linda	14-MAY-15	Elsternwick, Victoria, Australia	(null)	1
2	33401062	Tomoya	14-MAY-15	Melbourne, Victoria, Australia	(null)	1
3	33539920	Robert	17-MAY-15	AU	(null)	10
4	13751407	Bianca	31-MAR-14	Fitzroy, Victoria, Australia	(null)	1
5	35595639	Miles	11-JUN-15	Tamarama, New South Wales, Australia	(null)	0
6	35621973	Molly	12-JUN-15	Melbourne, Victoria, Australia	(null)	1
7	35748115	Bier4567	14-JUN-15	AU	(null)	1
8	26975890	Bob	31-JAN-15	Seville, Victoria, Australia	(null)	2
9	39573096	Jesse	26-JUL-15	Blackstone Heights, Tasmania, Australia	(null)	1
10	34493214	Mira	29-MAY-15	Victoria, Australia	(null)	1
11	39585816	Jacinta	26-JUL-15	Dalkeith, Western Australia, Australia	(null)	0
12	11400271	Kristen	17-JAN-14	AU	(null)	1
13	6354616	Simon	12-MAY-13	Melbourne, Victoria, Australia	(null)	1

Solution/Data cleaning SQL Code:

```
-- Solution
-- Update the NULL values with a default value (e.g., 'Unknown')
UPDATE HOST
SET Host_Location = 'Unknown'
WHERE Host_Location IS NULL;

UPDATE HOST
SET Host_About = 'No About'
WHERE Host_About IS NULL;

-- Check for NULL in HOST_LISTING_COUNT
SELECT *
FROM Host
WHERE HOST_LOCATION = 'Unknown';


SELECT *
FROM Host
WHERE HOST_About = 'No About';
```

Solution:

Although the primary function of the `Host_Location` column is to store the location of hosts an important piece of information the presence of `NULL` values undermines the purpose of the table. A missing location can hinder effective analysis, especially when location is key to understanding or evaluating host-related activities such as property availability or proximity for guests.

When faced with missing data, two main approaches were considered:

1. **Delete the rows with `NULL` values**:
    - This approach would ensure that the table only contains complete data, free from any missing values. However, this would also result in the permanent loss of all other potentially valuable data related to those hosts (e.g., host names, listing information).
    - This could significantly reduce the size and usability of the dataset, leading to gaps in host-related analysis.
2. **Replace the `NULL` values with `'Unknown'`**:
    - By replacing the `NULL` values with `'Unknown'`, we preserve all the other valuable data associated with the hosts. This also keeps the dataset complete without losing critical information about host properties, names, and activity, which could be crucial for business insights and reports.
    - While `'Unknown'` is not ideal, it allows us to maintain the integrity of the dataset while signalling that the location data is missing. This also gives the opportunity for future data enrichment if the host location can be identified later.

Why Replace with `'Unknown'` Instead of Deleting:

I chose to **replace the `NULL` values with `'Unknown'`** rather than deleting the records because:

- **Preservation of Data**: The host table contains more than just location data. By replacing `NULL` values, we avoid discarding valuable host-related information that can still be used for other analyses.
- **Signalling Missing Data**: Using `'Unknown'` allows us to mark the data as missing without losing track of the hosts entirely. This gives an opportunity to fill in the missing data in the future without the need to recover deleted records.
- **Completeness of Dataset**: Deleting rows with missing locations might leave gaps in the analysis, making it less comprehensive. Retaining the records with a placeholder value ensures the dataset remains robust, even if some data points are incomplete.

The `Host_About` column is designed to store descriptive information about the hosts. While this is not as critical as other fields, such as `Host_Location`, the presence of `NULL` values in this column can still undermine the completeness of the dataset. Even though the absence of this information might not significantly impact analysis, it could affect the richness of host profiles in certain reports or insights.

When faced with missing data in the Host_About column, two approaches were considered:

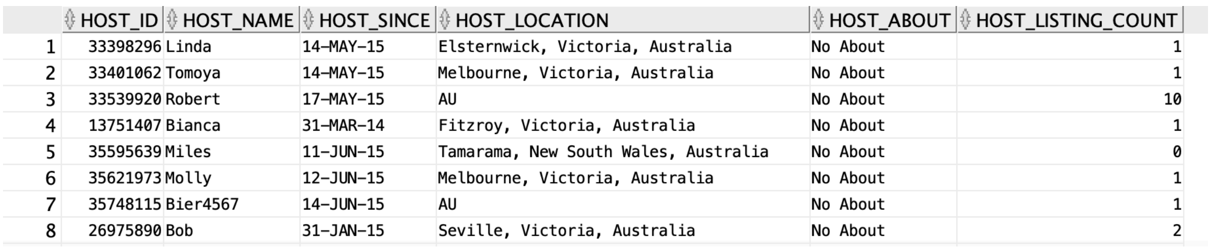
1. **Delete the rows with NULL values**:
   - This approach would ensure that the dataset only contains complete host profiles, free from any missing information. However, this would result in the permanent loss of potentially valuable host data, such as host IDs, names, or other key attributes.
   - Removing these rows would significantly reduce the size and usability of the dataset, especially when the Host_About column is not a crucial piece of information for most analyses.
2. **Replace the NULL values with 'No About'**:
   - By replacing the NULL values with a placeholder like 'No About', we retain all the valuable data associated with the hosts, such as host IDs, listings, and names. This also allows us to maintain a more complete dataset without the risk of losing important records, even if they lack descriptive details.
   - While 'No About' is not as informative as a real description, it signals that no data was provided for that host. This approach allows us to preserve the integrity of the dataset while maintaining the possibility of adding more complete information later if needed.

Why Replace with 'No About' Instead of Deleting:

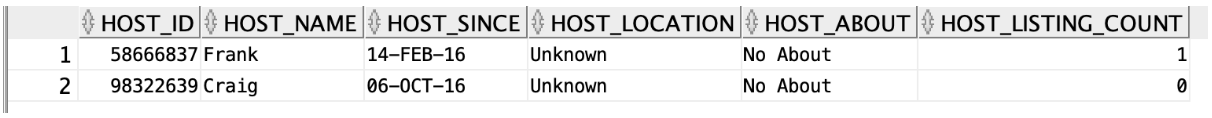
I chose to replace the NULL values in Host_About with 'No About' rather than deleting the records because:

- **Preservation of Data**: The host table contains essential information, and deleting rows due to missing descriptions would result in a loss of valuable data. By keeping the rows and replacing NULL values, we ensure that all other relevant host information remains intact and usable for analysis.
- **Signalling Missing Data**: Using 'No About' as a placeholder clearly indicates that no descriptive information was provided for those hosts. This allows analysts to understand which entries lack details without permanently losing track of these hosts.
- **Completeness of the Dataset**: Deleting rows with missing Host_About values would reduce the richness of the dataset, especially for analyses that do not depend on this column. By replacing NULL values with 'No About', we maintain the dataset's robustness, even if some hosts are missing descriptions.

The screenshot below shows the clean data



	HOST_ID	HOST_NAME	HOST_SINCE	HOST_LOCATION	HOST_ABOUT	HOST_LISTING_COUNT
1	33398296	Linda	14-MAY-15	Elsternwick, Victoria, Australia	No About	1
2	33401062	Tomoya	14-MAY-15	Melbourne, Victoria, Australia	No About	1
3	33539920	Robert	17-MAY-15	AU	No About	10
4	13751407	Bianca	31-MAR-14	Fitzroy, Victoria, Australia	No About	1
5	35595639	Miles	11-JUN-15	Tamarama, New South Wales, Australia	No About	0
6	35621973	Molly	12-JUN-15	Melbourne, Victoria, Australia	No About	1
7	35748115	Bier4567	14-JUN-15	AU	No About	1
8	26975890	Bob	31-JAN-15	Seville, Victoria, Australia	No About	2



	HOST_ID	HOST_NAME	HOST_SINCE	HOST_LOCATION	HOST_ABOUT	HOST_LISTING_COUNT
1	58666837	Frank	14-FEB-16	Unknown	No About	1
2	98322639	Craig	06-OCT-16	Unknown	No About	0

Strategy 4: Data Validation for Numeric Consistency

It involves checking numeric columns for **invalid or inconsistent values**, such as **negative numbers** where they are **logically incorrect** (e.g., listing_price, listing_max_nights, or booking_cost), to ensure data quality and integrity in the dataset.

4.1

In our exploration approach, the focus was on ensuring **data validation for numeric consistency**. This involved examining numeric columns across various tables, such as `listing_price`, `listing_max_nights`, and `booking_cost`, to identify any logically incorrect or inconsistent values. The aim was to detect errors like negative numbers in fields where such values are not meaningful, and to ensure overall data quality and integrity.

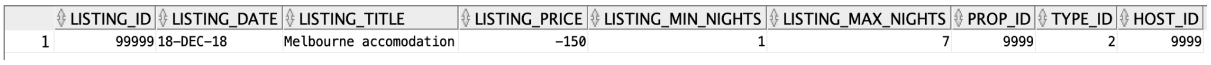
Exploration/Identifying Issue SQL Code:

```
 -- Select all listings where the listing price is negative (error: listing prices should not be negative)
SELECT *
FROM listing
WHERE listing_price < 0;   -- error: this identifies incorrect data (negative price)
--------------------------------------------------------------------------------
-- Check if there are any listings where the maximum number of nights is negative (error: max nights should not be negative)
SELECT *
FROM listing
WHERE listing_max_nights < 0;  -- This identifies any incorrect values for maximum nights

-- Recheck if there are still any listings with a negative listing price (should return no rows after correction)
SELECT *
FROM listing
WHERE listing_price < 0;  -- This ensures no listing has a negative price after the correction
----------------------------------------------------------------------------
SELECT *
FROM booking
WHERE booking_cost < 0;
```

Issue:

During the exploration, we identified an incorrect negative value in the `listing_price` field. Specifically, the `listing_price` for `listing_id = 99999` was recorded as `-150`, which is logically inconsistent, as prices should not be negative. Additionally, checks were performed on the `listing_max_nights` column to identify any negative values, as it is illogical to have a negative maximum number of nights. Similar checks were also conducted for the `booking_cost` field to ensure there were no negative values in fields representing monetary amounts.

You can refer to the screenshot attached below which shows the dirty data



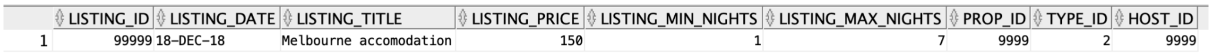
	LISTING_ID	LISTING_DATE	LISTING_TITLE	LISTING_PRICE	LISTING_MIN_NIGHTS	LISTING_MAX_NIGHTS	PROP_ID	TYPE_ID	HOST_ID
1	99999	18-DEC-18	Melbourne accomodation	-150	1	7	9999	2	9999

Solution SQL Code:

```
-- Solution: Correct the negative price for listing with listing_id = 99999
-- In this case, the value of -150 is corrected to 150 based on data analysis and comparison with similar listings.
SELECT * FROM listing;
-- Update the incorrect listing_price to a corrected value of 150 for the listing with listing_id = 99999
UPDATE listing
SET listing_price = 150
WHERE listing_id = 99999;

-- Verify that the data has been updated correctly by selecting all records from the listing table
SELECT *
FROM listing
WHERE listing_price < 0;
```

Solution:

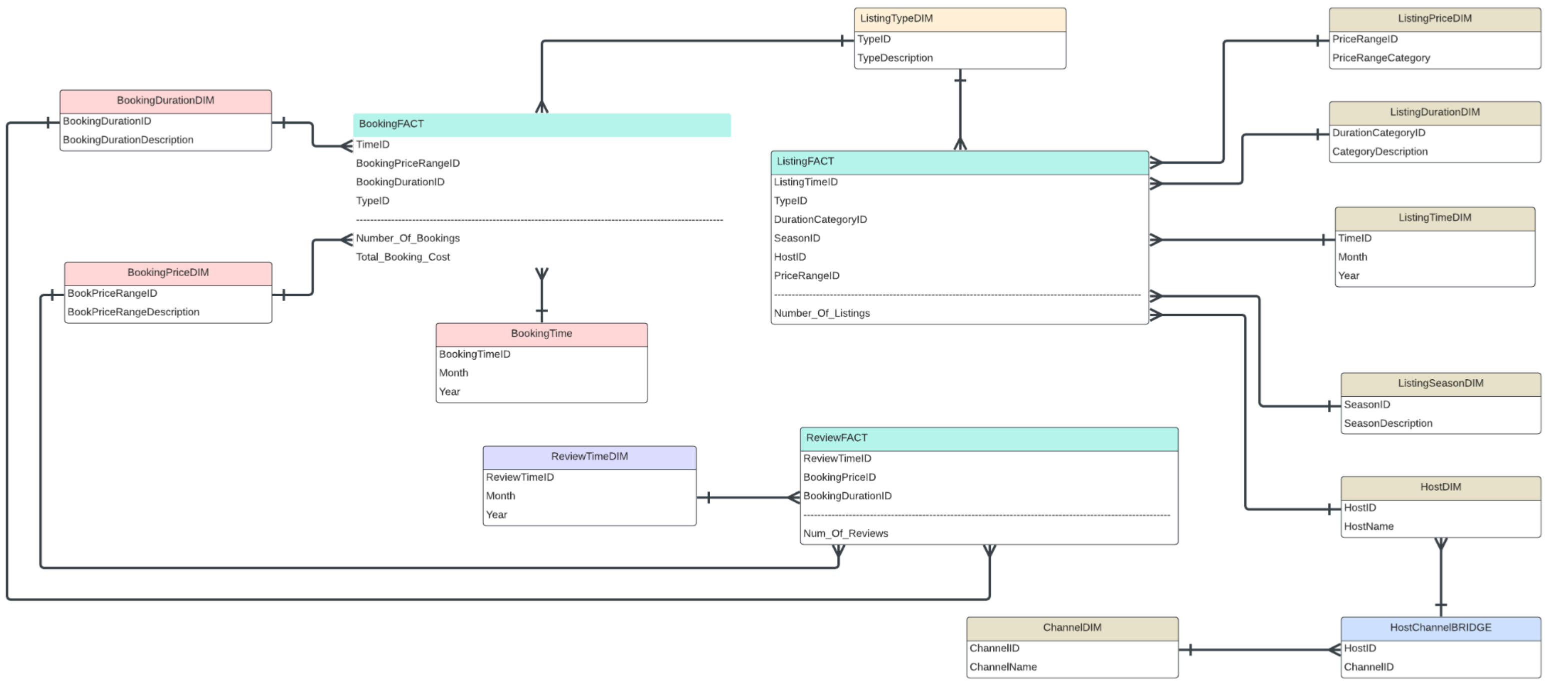
To address the issue, we explored the `listing` table and determined that the negative value for `listing_price` was likely a simple data entry error. Given the context of other listings and their price ranges, it was reasonable to conclude that the value of `-150` was incorrect. Therefore, we corrected this value to `150`, which aligns with similar listings and maintains data consistency. After updating the value for `listing_id = 99999`, we performed additional checks to ensure no other listings had negative prices.
You can refer to the screenshot attached below to see the clean data



	LISTING_ID	LISTING_DATE	LISTING_TITLE	LISTING_PRICE	LISTING_MIN_NIGHTS	LISTING_MAX_NIGHTS	PROP_ID	TYPE_ID	HOST_ID
1	99999	18-DEC-18	Melbourne accomodation	150	1	7	9999	2	9999

Star Schema Design and Granularity Suggestions

Star Schema:



Granularity Suggestions:

1. **Incorporate Individual Listing IDs into Fact Tables**

   By adding the *listing_id* to my *ListingFact* and *BookingFact* tables, I can analyse data at the individual listing level. This allows me to drill down into specific listings to evaluate their performance, occupancy rates, and revenue generation. It offers a granular view of which listings are the most popular or profitable.

2. **Add Daily Granularity to Time Dimensions**

   Currently, my time dimensions might be aggregated at the month or year level. Increasing the granularity to include day, or even hour, in my *BookingTimeDim*, *ListingTimeDim*, and *ReviewTimeDim* allows for a more detailed temporal analysis. This enables me to identify daily booking patterns, peak times, and understand how specific dates or events influence business performance.

3. **Include Guest Information**

   By integrating *guest_id* into my *BookingFact* table, I can analyse bookings at the customer level. This enhancement allows me to drill down into customer behaviours, preferences, and booking frequencies. It helps with crafting personalised marketing strategies and offers insights into customer lifetime value and retention rates.

4. **Incorporate Detailed Host Location Data**

   Enhancing my fact tables by adding more specific location attributes from host_location, such as city, neighbourhood, or *postal_code*, provides increased granularity for geographic analysis. This enables me to pinpoint high-demand areas, identify local market trends, and uncover opportunities for expansion or targeted promotions.

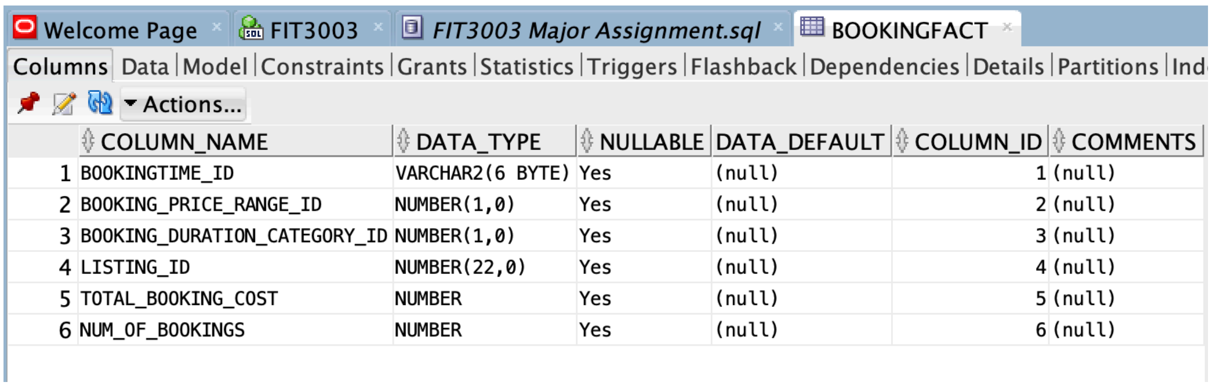
5. **Add Detailed Listing Attributes**

   By including additional listing attributes such as *room_type*, *number_of_bedrooms*, *amenities and my fact* tables, I can perform a more nuanced analysis of how different property features impact performance metrics, like booking rates and or listing prices. This allows me to assess the popularity and profitability of various property types and features in greater detail.

Star Schema Design (Screenshots)

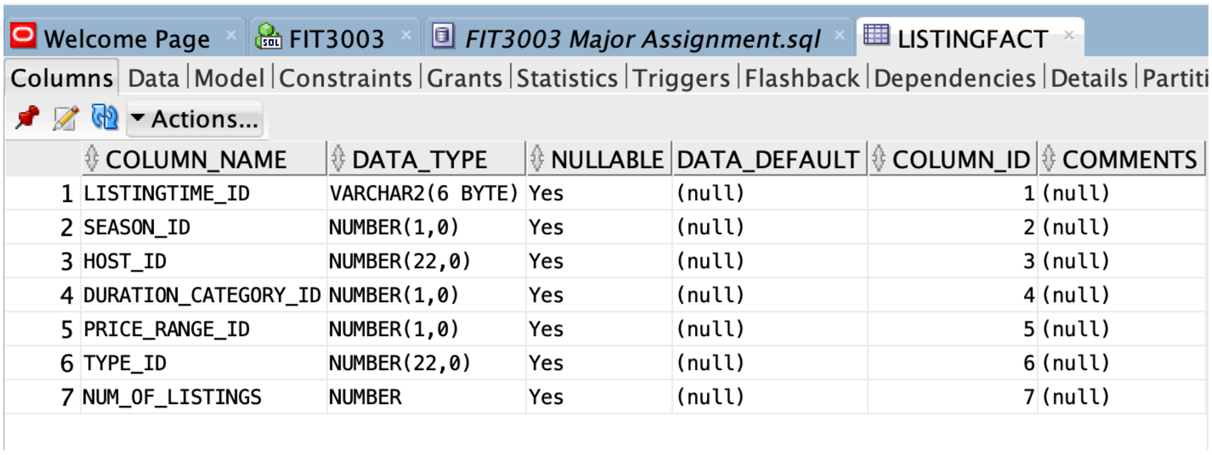
Fact Tables:

BookingFact


Welcome Page
FIT3003
FIT3003 Major Assignment.sql
BOOKINGFACT
Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Ind
Actions...

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	BOOKINGTIME_ID	VARCHAR2(6 BYTE)	Yes	(null)	1	(null)
2	BOOKING_PRICE_RANGE_ID	NUMBER(1,0)	Yes	(null)	2	(null)
3	BOOKING_DURATION_CATEGORY_ID	NUMBER(1,0)	Yes	(null)	3	(null)
4	LISTING_ID	NUMBER(22,0)	Yes	(null)	4	(null)
5	TOTAL_BOOKING_COST	NUMBER	Yes	(null)	5	(null)
6	NUM_OF_BOOKINGS	NUMBER	Yes	(null)	6	(null)

ListingFact



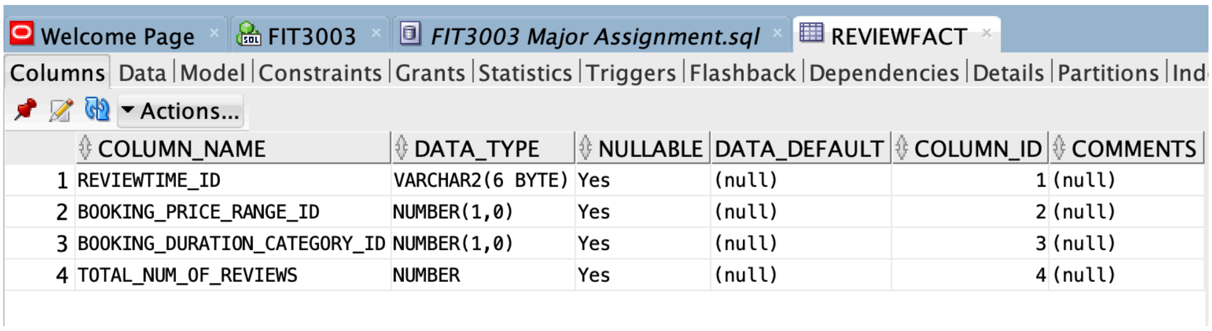
Welcome Page | FIT3003 | *FIT3003 Major Assignment.sql* | LISTINGFACT

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partiti

Actions...

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	LISTINGTIME_ID	VARCHAR2(6 BYTE)	Yes	(null)	1	(null)
2	SEASON_ID	NUMBER(1,0)	Yes	(null)	2	(null)
3	HOST_ID	NUMBER(22,0)	Yes	(null)	3	(null)
4	DURATION_CATEGORY_ID	NUMBER(1,0)	Yes	(null)	4	(null)
5	PRICE_RANGE_ID	NUMBER(1,0)	Yes	(null)	5	(null)
6	TYPE_ID	NUMBER(22,0)	Yes	(null)	6	(null)
7	NUM_OF_LISTINGS	NUMBER	Yes	(null)	7	(null)

ReviewFact



Welcome Page | FIT3003 | *FIT3003 Major Assignment.sql* | REVIEWFACT

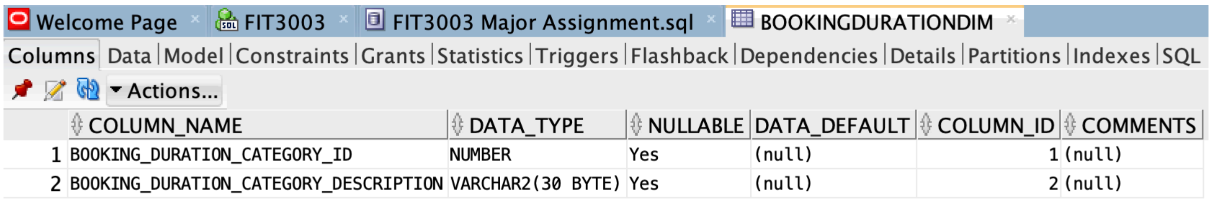
Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Ind

Actions...

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	REVIEWTIME_ID	VARCHAR2(6 BYTE)	Yes	(null)	1	(null)
2	BOOKING_PRICE_RANGE_ID	NUMBER(1,0)	Yes	(null)	2	(null)
3	BOOKING_DURATION_CATEGORY_ID	NUMBER(1,0)	Yes	(null)	3	(null)
4	TOTAL_NUM_OF_REVIEWS	NUMBER	Yes	(null)	4	(null)

Dimension Tables:

BookingDurationDIM



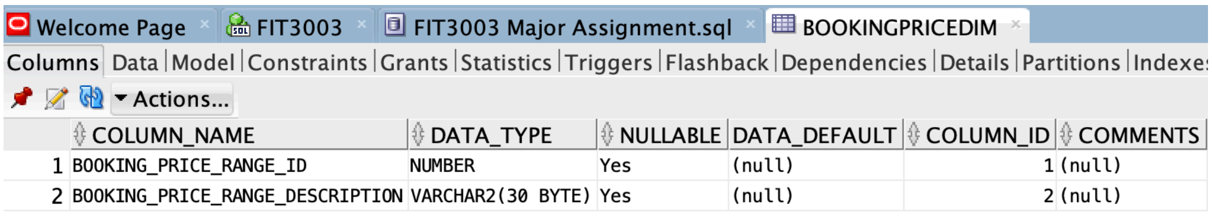
Welcome Page | FIT3003 | FIT3003 Major Assignment.sql | BOOKINGDURATIONDIM

Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partitions | Indexes | SQL

Actions...

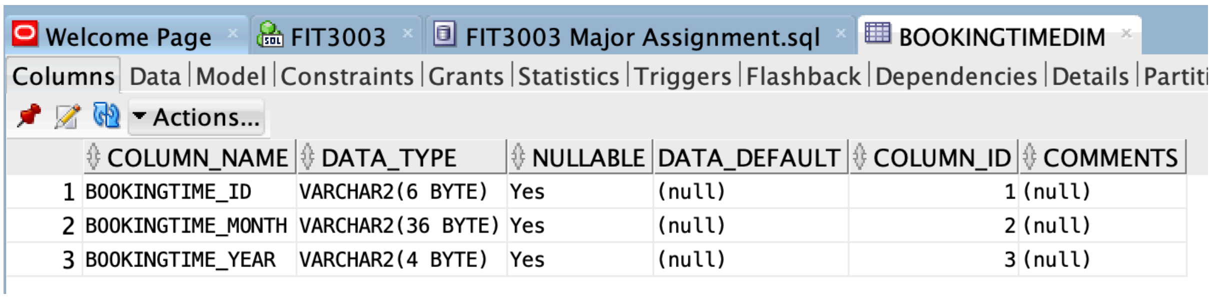
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	BOOKING_DURATION_CATEGORY_ID	NUMBER	Yes	(null)	1	(null)
2	BOOKING_DURATION_CATEGORY_DESCRIPTION	VARCHAR2(30 BYTE)	Yes	(null)	2	(null)

BookingPriceRangeDIM



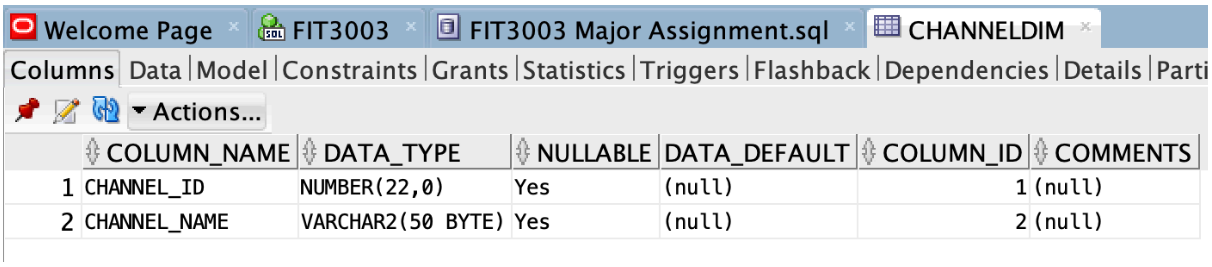
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	BOOKING_PRICE_RANGE_ID	NUMBER	Yes	(null)	1	(null)
2	BOOKING_PRICE_RANGE_DESCRIPTION	VARCHAR2(30 BYTE)	Yes	(null)	2	(null)

BookingTimeDIM



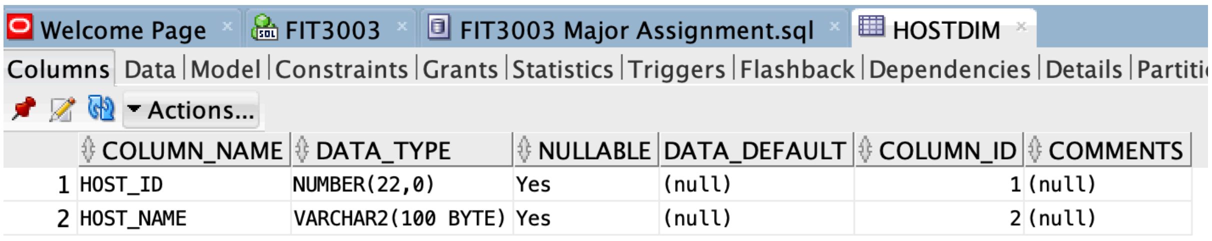
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	BOOKINGTIME_ID	VARCHAR2(6 BYTE)	Yes	(null)	1	(null)
2	BOOKINGTIME_MONTH	VARCHAR2(36 BYTE)	Yes	(null)	2	(null)
3	BOOKINGTIME_YEAR	VARCHAR2(4 BYTE)	Yes	(null)	3	(null)

ChannelDIM



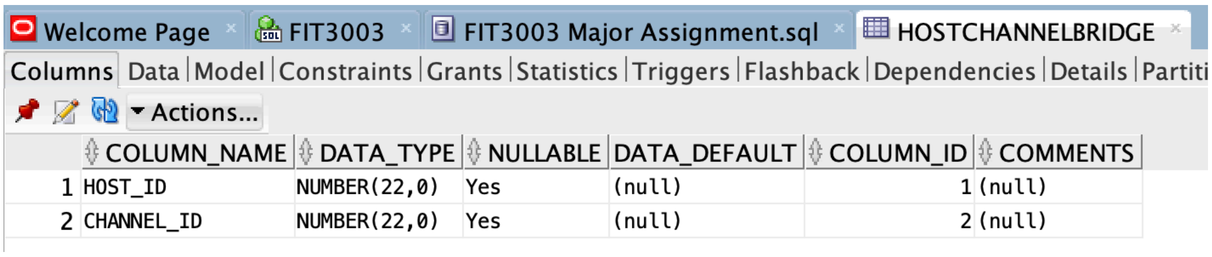
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	CHANNEL_ID	NUMBER(22,0)	Yes	(null)	1	(null)
2	CHANNEL_NAME	VARCHAR2(50 BYTE)	Yes	(null)	2	(null)

HostDIM



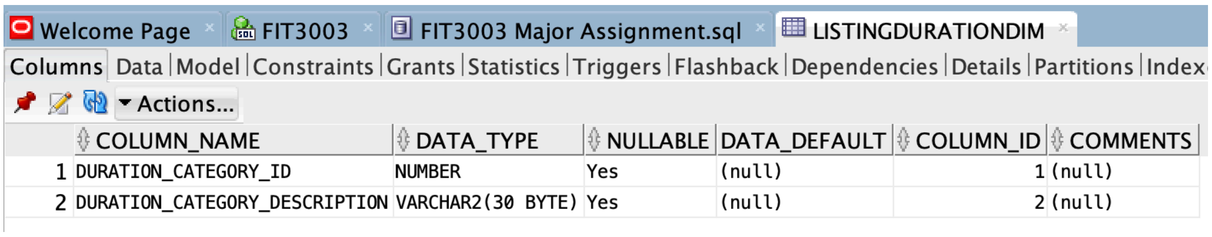
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	HOST_ID	NUMBER(22,0)	Yes	(null)	1	(null)
2	HOST_NAME	VARCHAR2(100 BYTE)	Yes	(null)	2	(null)

HostChannelBridge



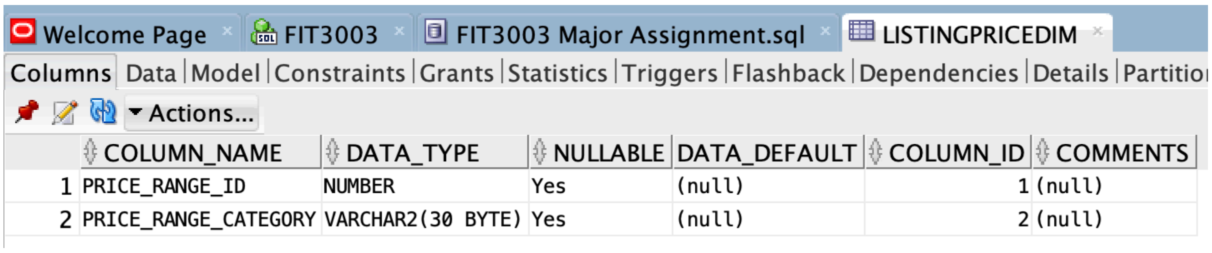
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	HOST_ID	NUMBER(22,0)	Yes	(null)	1	(null)
2	CHANNEL_ID	NUMBER(22,0)	Yes	(null)	2	(null)

ListingDurationDIM


Welcome Page
FIT3003
FIT3003 Major Assignment.sql
LISTINGDURATIONDIM
Columns Data Model Constraints Grants Statistics Triggers Flashback Dependencies Details Partitions Index
Actions...

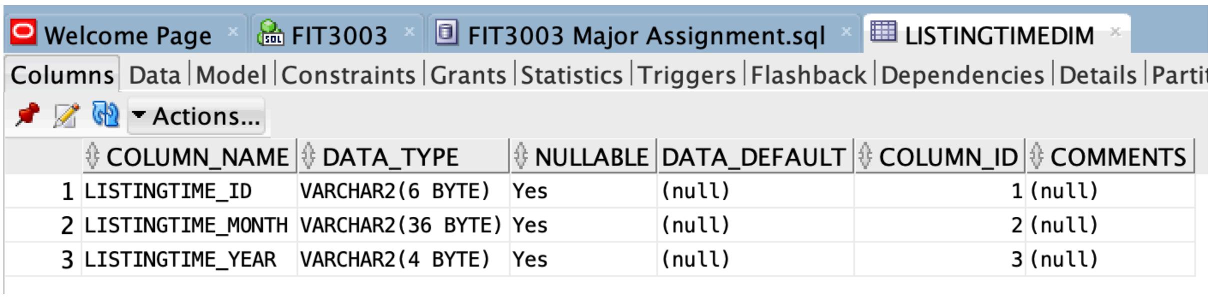
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	DURATION_CATEGORY_ID	NUMBER	Yes	(null)	1	(null)
2	DURATION_CATEGORY_DESCRIPTION	VARCHAR2(30 BYTE)	Yes	(null)	2	(null)

ListingPriceDIM


Welcome Page
FIT3003
FIT3003 Major Assignment.sql
LISTINGPRICEDIM
Columns Data Model Constraints Grants Statistics Triggers Flashback Dependencies Details Partitio
Actions...

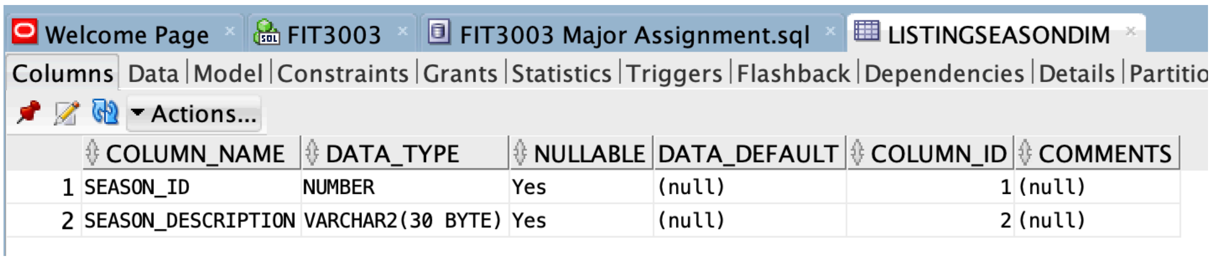
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	PRICE_RANGE_ID	NUMBER	Yes	(null)	1	(null)
2	PRICE_RANGE_CATEGORY	VARCHAR2(30 BYTE)	Yes	(null)	2	(null)

ListingTimeDIM


Welcome Page
FIT3003
FIT3003 Major Assignment.sql
LISTINGTIMEDIM
Columns Data Model Constraints Grants Statistics Triggers Flashback Dependencies Details Parti
Actions...

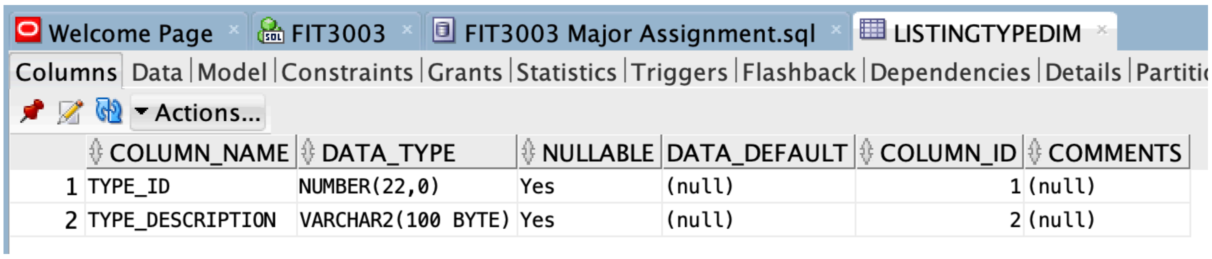
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	LISTINGTIME_ID	VARCHAR2(6 BYTE)	Yes	(null)	1	(null)
2	LISTINGTIME_MONTH	VARCHAR2(36 BYTE)	Yes	(null)	2	(null)
3	LISTINGTIME_YEAR	VARCHAR2(4 BYTE)	Yes	(null)	3	(null)

ListingSeasonDIM


Welcome Page
FIT3003
FIT3003 Major Assignment.sql
LISTINGSEASONDIM
Columns Data Model Constraints Grants Statistics Triggers Flashback Dependencies Details Partitio
Actions...

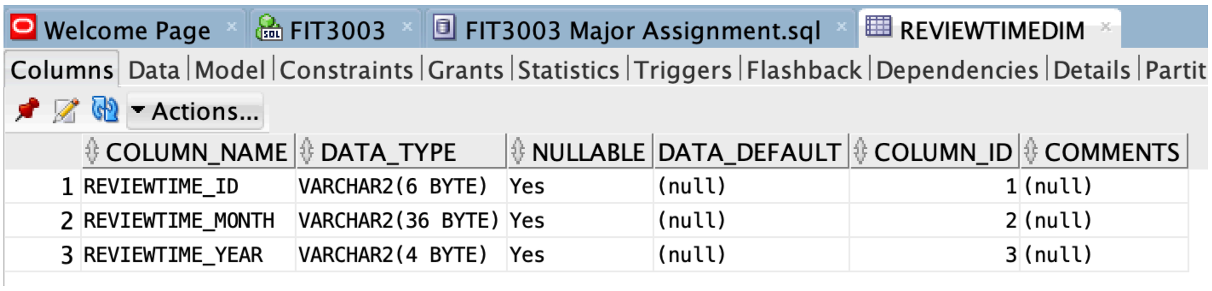
	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	SEASON_ID	NUMBER	Yes	(null)	1	(null)
2	SEASON_DESCRIPTION	VARCHAR2(30 BYTE)	Yes	(null)	2	(null)

ListingTypeDIM


Welcome Page
FIT3003
FIT3003 Major Assignment.sql
LISTINGTYPEDIM
Columns Data Model Constraints Grants Statistics Triggers Flashback Dependencies Details Partitic
Actions...

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	TYPE_ID	NUMBER(22,0)	Yes	(null)	1	(null)
2	TYPE_DESCRIPTION	VARCHAR2(100 BYTE)	Yes	(null)	2	(null)

ReviewTimeDIM


Welcome Page
FIT3003
FIT3003 Major Assignment.sql
REVIEWTIMEDIM
Columns | Data | Model | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Partit
Actions...

	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	REVIEWTIME_ID	VARCHAR2(6 BYTE)	Yes	(null)	1	(null)
2	REVIEWTIME_MONTH	VARCHAR2(36 BYTE)	Yes	(null)	2	(null)
3	REVIEWTIME_YEAR	VARCHAR2(4 BYTE)	Yes	(null)	3	(null)

Data Analytic Stage

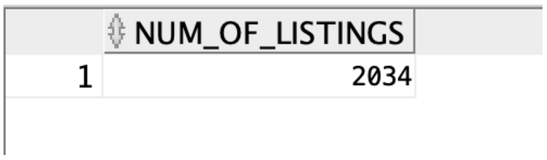
Query 1: How Many Long-Term Stay Duration Listings Are Listed on Facebook?

SQL Code:

To determine the number of long-term stay duration listings available on Facebook, the following SQL query was executed:

```
SELECT SUM(L.num_of_listings) AS num_of_listings FROM ListingFact L
JOIN HOSTCHANNELBRIDGE HV ON L.host_id = HV.host_id JOIN channeldim
C ON HV.channel_id = C.channel_id JOIN LISTINGDURATIONDIM D ON
L.duration_category_id = D.DURATION_CATEGORY_ID WHERE
UPPER(D.DURATION_CATEGORY_DESCRIPTION) = 'LONG-TERM' AND
UPPER(C.channel_name) = 'FACEBOOK';
```

Screenshot Attached Below for Results



	NUM_OF_LISTINGS
1	2034

Findings:

The analysis reveals that there are **2,034 long-term stay duration listings** available on Facebook. This indicates that Facebook is a significant platform for hosting long-term stays, reflecting its potential popularity among hosts offering such services.

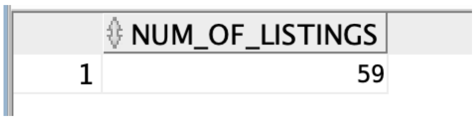
Query 2: How Many Listings Are Listed in June 2015?

SQL Code:

To find out the number of listings available in June 2015, the following SQL query was used:

```
SELECT SUM(num_of_listings) AS num_of_listings FROM ListingFact
WHERE ListingTime_ID = '201506';
```

Screenshot Attached Below for Results



	NUM_OF_LISTINGS
1	59

Findings:

In June 2015, a total of **59 listings** were available. This could reflect a seasonal pattern or other external factors influencing the number of listings at that time.

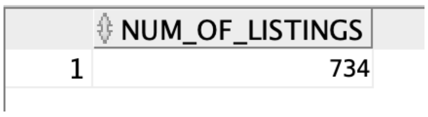
Query 3: How Many Listings Are There in Summer for an “Entire Home/Apt” in a Medium Price Range?

SQL Code:

To determine the number of listings during the summer season for entire homes or apartments in the medium price range, the following SQL query was executed:

```
SELECT SUM(L.num_of_listings) AS num_of_listings FROM ListingFact L
JOIN LISTINGSEASONDIM S ON L.season_id = S.Season_ID JOIN
ListingTypeDIM T ON L.type_id = T.type_id JOIN LISTINGPRICEDIM P ON
L.price_range_id = P.price_range_id WHERE S.Season_Description =
'Summer' AND T.Type_DESCRIPTION = 'Entire home/apt' AND
P.PRICE_RANGE_CATEGORY = 'Medium';
```

Screenshot Attached Below for Results



	NUM_OF_LISTINGS
1	734

Findings:

There were **734 listings** for entire homes or apartments in the medium price range during the summer season. This suggests that medium-priced listings are quite popular for entire homes/apartments, especially during summer, which is often a high-demand travel period.

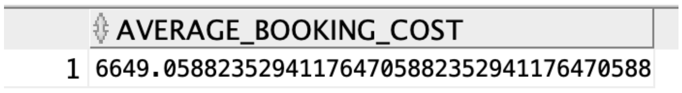
Query 4: How Much Is the Average Booking Cost in March 2013?

SQL Code:

To calculate the average booking cost in March 2013, the following SQL query was executed:

```
SELECT (total_booking_cost / num_of_bookings) AS
average_booking_cost FROM BookingFact WHERE BookingTime_ID =
'201303';
```

Screenshot Attached Below for Results



	AVERAGE_BOOKING_COST
1	6649.0588235294117647058823529411764705 88

Findings:

The average booking cost in March 2013 was **6,649.06**. This relatively high average cost might reflect the popularity of certain locations or the inclusion of high-end listings during that period. Comparing this with the costs in other months or years would help identify trends.

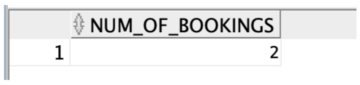
Query 5: How Many Bookings Were There for “Private Rooms” with a Short-Term Stay Duration in 2015?

SQL Code:

To determine the number of bookings for private rooms with a short-term stay duration in 2015, the following SQL query was executed:

```
SELECT SUM(B.num_of_bookings) AS num_of_bookings FROM bookingfact B
JOIN BOOKINGTIMEDIM BT ON B.bookingtime_id = BT.BookingTime_ID JOIN
BOOKINGDURATIONDIM D ON B.BOOKING_DURATION_CATEGORY_ID =
D.BOOKING_DURATION_CATEGORY_ID JOIN Listing L ON B.listing_id =
L.listing_id JOIN ListingTypeDIM T ON L.type_id = T.type_id WHERE
D.BOOKING_DURATION_CATEGORY_DESCRIPTION = 'Short-term' AND
T.Type_Description = 'Private room' AND BT.BookingTime_Year =
'2015';
```

Screenshot Attached Below for Results



	NUM_OF_BOOKINGS
1	2

Findings:

A total of **2 bookings** for private rooms with short-term stay duration were made in 2015. This low number could indicate either a niche market for short-term private room rentals or could reflect a particular downturn during this year. More context on overall trends for private room bookings would clarify this.

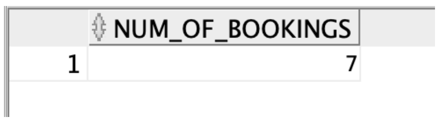
Query 6: How Many High-Cost Bookings Were Made in April 2014?

SQL Code:

To find out the number of high-cost bookings made in April 2014, the following SQL query was used:

```
SELECT SUM(num_of_bookings) AS num_of_bookings FROM BookingFact
WHERE Booking_Price_Range_ID = 3 AND BookingTime_ID = '201404';
```

Screenshot Attached Below for Results



	NUM_OF_BOOKINGS
1	7

Findings:

There were **7 high-cost bookings** made in April 2014. This suggests that high-cost bookings, while relatively few in number, may still cater to a specific market segment, potentially indicating luxury or high-end rentals.

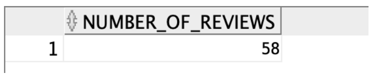
Query 7: How Many Reviews Were Given in February 2016?

SQL Code:

To determine the number of reviews given in February 2016, the following SQL query was executed:

```
SELECT SUM(total_num_of_reviews) AS number_of_reviews FROM
ReviewFact WHERE ReviewTime_ID = '201602';
```

Screenshot Attached Below for Results



	NUMBER_OF_REVIEWS
1	58

Findings:

In February 2016, **58 reviews** were submitted. This provides an indication of customer engagement, reflecting both satisfaction and interaction with listings during that month.

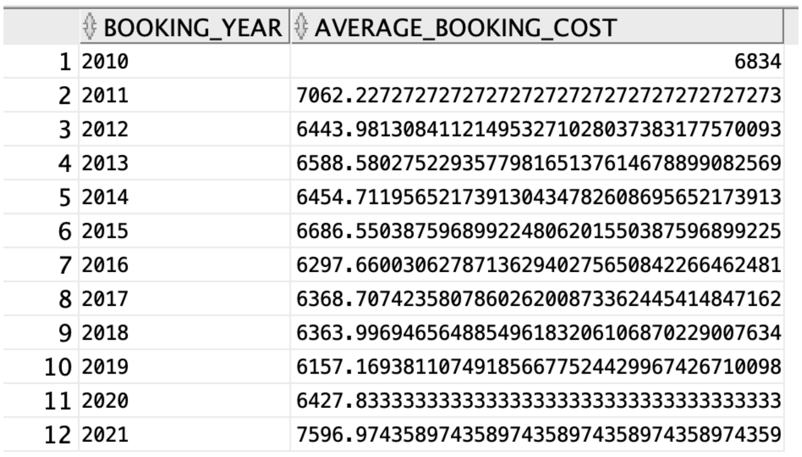
Query 8: What Is the Average Booking Cost Per Year?

SQL Code:

To calculate the average booking cost per year, the following SQL query was executed:

```
SELECT BT.BookingTime_Year AS Booking_Year,
SUM(BF.total_booking_cost) / SUM(BF.num_of_bookings) AS
average_booking_cost FROM BookingFact BF JOIN BOOKINGTIMEDIM BT ON
BF.bookingtime_id = BT.BookingTime_ID GROUP BY BT.BookingTime_Year
ORDER BY BT.BookingTime_Year;
```

Screenshot Attached Below for Results



	BOOKING_YEAR	AVERAGE_BOOKING_COST
1	2010	6834
2	2011	7062.227272727272727272727272727272727273
3	2012	6443.981308411214953271028037383177570093
4	2013	6588.580275229357798165137614678899082569
5	2014	6454.711956521739130434782608695652173913
6	2015	6686.550387596899224806201550387596899225
7	2016	6297.660030627871362940275650842266462481
8	2017	6368.707423580786026200873362445414847162
9	2018	6363.996946564885496183206106870229007634
10	2019	6157.169381107491856677524429967426710098
11	2020	6427.833333333333333333333333333333333333
12	2021	7596.974358974358974358974358974358974359

Findings:

The average booking cost fluctuated over the years, with the lowest in 2019 at **6,157.17** and the highest in 2021 at **7,596.97**. The consistent rise post-2019 could indicate inflation or growing demand for high-value bookings, with a significant spike in 2021 possibly reflecting post-pandemic travel resurgence.

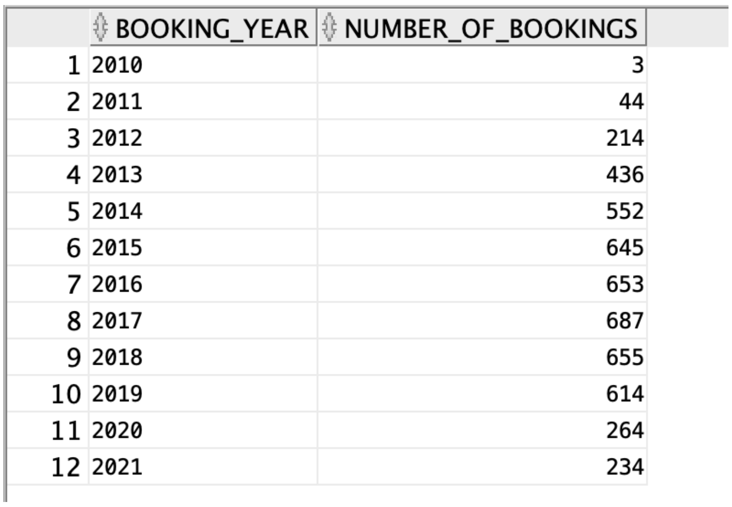
Query 9: How Many Bookings Are There Per Year?

SQL Code:

To determine the number of bookings per year, the following SQL query was executed:

```
SELECT BT.BookingTime_Year AS Booking_Year, SUM(BF.num_of_bookings)
AS number_of_bookings FROM BookingFact BF JOIN BOOKINGTIMEDIM BT ON
BF.bookingtime_id = BT.BookingTime_ID GROUP BY BT.BookingTime_Year
ORDER BY BT.BookingTime_Year;
```

Screenshot Attached Below for Results



	BOOKING_YEAR	NUMBER_OF_BOOKINGS
1	2010	3
2	2011	44
3	2012	214
4	2013	436
5	2014	552
6	2015	645
7	2016	653
8	2017	687
9	2018	655
10	2019	614
11	2020	264
12	2021	234

Findings:

The number of bookings varied significantly across the years. From just **3 bookings in 2010** to **687 in 2017**, the data reflects a general upward trend, suggesting growth in the platform's usage and popularity. However, there was a notable drop in 2020 and 2021, with **264 and 234 bookings** respectively, likely reflecting the global travel impact caused by the pandemic.

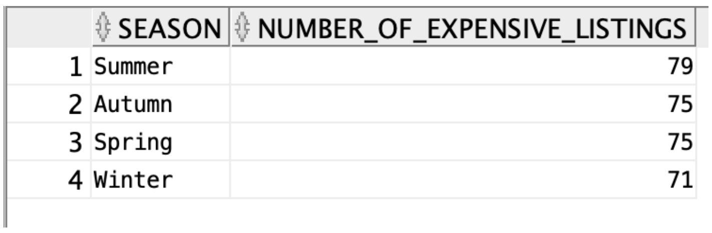
Query 10: Which Season Had the Most Expensive Listings?

SQL Code:

To identify which season had the most expensive listings, the following SQL query was executed:

```
SELECT S.Season_Description AS Season, SUM(LF.num_of_listings) AS
number_of_expensive_listings FROM ListingFact LF JOIN
LISTINGSEASONDIM S ON LF.season_id = S.Season_ID WHERE
LF.price_range_id = 3 -- Filter for the most expensive listings
GROUP BY S.Season_Description ORDER BY number_of_expensive_listings
DESC;
```

Screenshot Attached Below for Results



	SEASON	NUMBER_OF_EXPENSIVE_LISTINGS
1	Summer	79
2	Autumn	75
3	Spring	75
4	Winter	71

Findings:

The **summer** season had the highest number of expensive listings at **79**, followed closely by autumn and spring at **75** each. This highlights summer as a peak season for luxury or high-end listings, possibly due to increased vacation and travel demand.

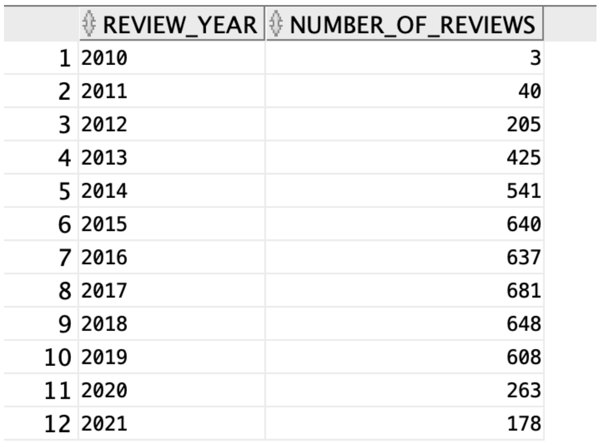
Query 11: How Many Reviews Were Given Each Year?

SQL Code:

To determine the number of reviews given each year, the following SQL query was executed:

```
SELECT RT.ReviewTime_Year AS Review_Year,
SUM(RF.total_num_of_reviews) AS number_of_reviews FROM ReviewFact RF
JOIN REVIEWTIMEDIM RT ON RF.ReviewTime_ID = RT.ReviewTime_ID GROUP
BY RT.ReviewTime_Year ORDER BY RT.ReviewTime_Year;
```

Screenshot Attached Below for Results



	REVIEW_YEAR	NUMBER_OF_REVIEWS
1	2010	3
2	2011	40
3	2012	205
4	2013	425
5	2014	541
6	2015	640
7	2016	637
8	2017	681
9	2018	648
10	2019	608
11	2020	263
12	2021	178

Findings:

The number of reviews has shown significant growth from **3 reviews in 2010** to **681 in 2017**, indicating increasing user engagement over the years. The peak was reached in 2017, after which there was a steady decline, with **263 reviews in 2020** and **178 in 2021**. This decrease aligns with the onset of the COVID-19 pandemic, which likely caused a downturn in travel, leading to fewer bookings and reviews.

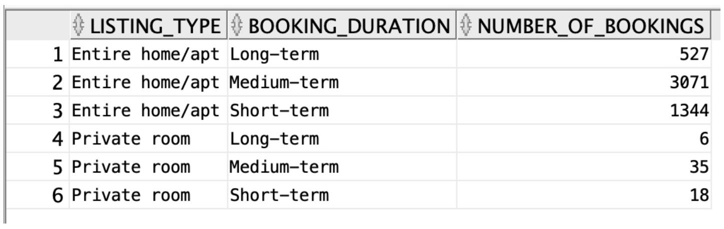
Query 12: Bookings by Listing Type and Booking Duration

SQL Code:

To analyze bookings by listing type and booking duration, the following SQL query was executed:

```
SELECT T.Type_Description AS Listing_Type,
D.BOOKING_DURATION_CATEGORY_DESCRIPTION AS Booking_Duration,
SUM(BF.num_of_bookings) AS Number_of_Bookings FROM BookingFact BF
JOIN BOOKINGDURATIONDIM D ON BF.BOOKING_DURATION_CATEGORY_ID =
D.BOOKING_DURATION_CATEGORY_ID JOIN Listing L ON BF.listing_id =
L.listing_id JOIN ListingTypeDIM T ON L.type_id = T.type_id GROUP BY
T.Type_Description, D.BOOKING_DURATION_CATEGORY_DESCRIPTION ORDER BY
T.Type_Description, D.BOOKING_DURATION_CATEGORY_DESCRIPTION;
```

Screenshot Attached Below for Results



	LISTING_TYPE	BOOKING_DURATION	NUMBER_OF_BOOKINGS
1	Entire home/apt	Long-term	527
2	Entire home/apt	Medium-term	3071
3	Entire home/apt	Short-term	1344
4	Private room	Long-term	6
5	Private room	Medium-term	35
6	Private room	Short-term	18

Findings:

The analysis shows that **entire homes/apartments dominate the bookings across all durations**, with **3,071 bookings for medium-term stays**, followed by **1,344 for short-term stays** and **527 for long-term stays**. In contrast, private rooms saw significantly fewer bookings, with only **35 medium-term**, **18 short-term**, and **6 long-term bookings**.

Pattern Analysis:

Entire homes/apartments are clearly more popular than private rooms, especially for medium-term stays. This suggests that travelers prefer the privacy and independence of an entire home, particularly for extended stays. The lower numbers for private rooms indicate that this option may cater to a more niche market, possibly budget travelers or those seeking shorter stays.

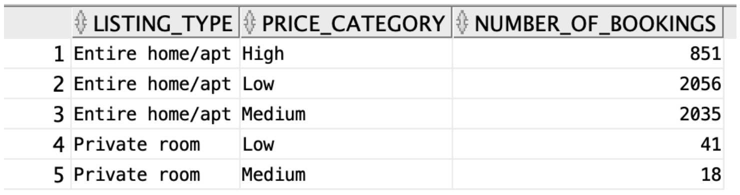
Query 13: Bookings by Listing Type and Price Category

SQL Code:

To analyze bookings by listing type and price category, the following SQL query was executed:

```
SELECT T.Type_Description AS Listing_Type,
D.BOOKING_Price_Range_DESCRIPTION AS Price_Category,
SUM(BF.num_of_bookings) AS Number_of_Bookings FROM BookingFact BF
JOIN BOOKINGPRICEDIM D ON BF.BOOKING_PRICE_RANGE_ID =
D.BOOKING_PRICE_RANGE_ID JOIN Listing L ON BF.listing_id =
L.listing_id JOIN ListingTypeDIM T ON L.type_id = T.type_id GROUP BY
T.Type_Description, D.BOOKING_Price_Range_DESCRIPTION ORDER BY
T.Type_Description, D.BOOKING_Price_Range_DESCRIPTION;
```

Screenshot Attached Below for Results



	LISTING_TYPE	PRICE_CATEGORY	NUMBER_OF_BOOKINGS
1	Entire home/apt	High	851
2	Entire home/apt	Low	2056
3	Entire home/apt	Medium	2035
4	Private room	Low	41
5	Private room	Medium	18

Findings:

The majority of bookings for entire homes/apartments are in the **low (2,056)** and **medium (2,035)** price categories, with only **851 bookings in the high-price range**. For private rooms, bookings are minimal across all categories, with **41 low-price** and **18 medium-price bookings**.

Pattern Analysis:

The data indicates that **low and medium price ranges are the most popular for entire homes/apartments**, suggesting that travelers on this platform are price-sensitive and prioritize affordability. The relatively low number of high-price bookings suggests that luxury listings cater to a smaller market segment. Similarly, private rooms, even in lower price ranges, attract fewer bookings, reinforcing the preference for entire homes/apartments.

Overall Pattern Analysis

The steady growth in reviews between **2010 and 2017** suggests increasing platform adoption and user satisfaction, as reviews often correlate with engagement and feedback. The sharp decline in **2020 and 2021** is consistent with the global travel restrictions during the pandemic, pointing to external factors significantly impacting user activity on the platform.

Entire homes/apartments are consistently more popular than private rooms, especially in the **medium-term stay** category, indicating a preference for privacy and independence among travelers. The **low and medium price ranges** dominate bookings, highlighting price sensitivity among users. **High-cost bookings**, while fewer in number, indicate a niche market for luxury or high-end rentals.

Seasonally, **summer** emerges as the peak period for both overall and high-end listings, likely driven by increased vacation and travel demand during this time. The data reflects broader trends in travel behavior, platform growth, and the impact of global events such as the COVID-19 pandemic on booking and review activities.