Querying and Aggregating Individual Tables with Google BigQuery

As we work through these examples we will be using many tables from the MIMIC-III demo dataset. You can find data dictionaries that explain what is in each tables in more detail on the MIMIC-III website.

Selecting All Data

If you want to select all columns and rows from a table you need to use the "*" in your select statement. Note that in Google BigQuery (and in most systems that have multiple datasets/databases available) you will need to include the dataset name before you call the table name. In this case the MIMIC-III demo data is stored in the dataset named "mimic3_demo". To query the PATIENTS table you can copy/paste the following code:

```
SELECT * FROM mimic3_demo.PATIENTS
```

You may get a warning about being charged for querying all the data in this data set. This is ok, remember that all of the charges for your queries of data in this project are being supported by our Industry Partner Google Cloud. (Note that if you were to query non course data sets your Google account would be billed for those queries.)

Selecting Variables (Columns) and Records (Rows)

If we only want to select the patient identifier and gender from the PATIENTS table we can run the following query:

```
SELECT SUBJECT_ID, GENDER FROM mimic3_demo.PATIENTS
```

Notice that this query returned a result for all 100 patients. If we run the same query but use "where" to only return patients who are women:

```
SELECT SUBJECT_ID, GENDER FROM mimic3_demo.PATIENTS where GENDER="F"
```

You should find that only 55 records are returned. You may be wondering, where am I finding these numbers? Google BigQuery returns results across multiple pages, you can navigate through them at the bottom of the table.

Techniques for Filtering Records

Up to now we have assumed you know what data is in a column and you know exactly what you want to select. Far more commonly in data science we know which variable we want to use to filter, but we're not sure the format of the data in the column. In addition to looking at the whole table by hand with the Preview option, there are two tools you can use:

Identify All Unique Variable Values

If the variable has some level of structure or format, then getting a list of all unique values of the variable makes it much easier to review. In traditional SQL we would use the "distinct" argument. So a generic query would be something like:

```
SELECT distinct gender FROM Patients
```

Unfortunately Google BigQuery only allows the distinct argument when applying functions (we'll see this later). To get a list of unique items, instead you will want to use the "group by" syntax.

Let's look at ADMISSIONS table and the ADMISSION_LOCATION variable. If we wanted to see all possible places the patients were admitted from, we can use the following query:

```
SELECT ADMISSION_LOCATION FROM mimic3_demo.ADMISSIONS

   GROUP BY ADMISSION_LOCATION
```

When you run this query you can see that there are only five unique places for admission location. You can then use the values of interest in your select query in the future!

Try it out for yourself: Using the ICUSTAYS table, how many types of FIRST_CAREUNIT are there?

Search the Variable Using "Like"

Although the previous technique is powerful, in some cases the variable will actually contain unstructured data or have too many options to make manually reviewing all distinct values unfeasible. In this case you can look for keywords or fuzzy matches with the syntax "like". This works just like a "where" clause, but instead of an equals sign, you use the "like" command to get more complicated matches.

Let's look at the D_ICD_DIAGNOSES table and apply the unique value approach:

```
SELECT LONG_TITLE FROM mimic3_demo.D_ICD_DIAGNOSES

   GROUP BY LONG_TITLE
```

You will find that there are more than 14,000 unique values for LONG_TITLE. If our goal was to find all unique entries that had something to do with diabetes, reading through all 14,562 options would take forever. Instead let's try the following query:

```
SELECT LONG_TITLE FROM mimic3_demo.D_ICD_DIAGNOSES

 WHERE lower(LONG_TITLE) like "%diabetes%"

 GROUP BY LONG_TITLE
```

Let's look a bit closer at this query. I have added a "where" statement on the second line. In this statement I use the lower() function to make data in LONG_TITLE all lower case - this is to make sure that capitalization doesn't matter. Then I use the "like" clause to say what the variable should match. In quotes I put my term "diabetes" and surround it with two percent (%) signs. In BigQuery the % symbol means that it can match any character. So this would catch any statement that has something before or after the word diabetes. When you run this query you'll see that it matched things where diabetes was the first word, somewhere in the middle, or at the end. Here are just a couple of illustrative examples of matches found:

Diabetes with other specified manifestations, type II or unspecified type, uncontrolled
Secondary diabetes mellitus with neurological manifestations, uncontrolled
Polyneuropathy in diabetes

Try it out for yourself: Using the D_ITEMS table, how many values in LABEL have the word "weight" in them?

Aggregating Data

Although much of our Specialization uses R to aggregate and analyze data, especially when you are working with large data it is more efficient to do some of the data aggregation in SQL.

SQL Functions (e.g., count(), min(), max(), etc.)

SQL functions are tools you can apply to your column select statement to get transformed variables. The most simple example is count(). This function counts the number of records or rows that are returned. For example if we wanted to know how many entries were in the ADMISSIONS table we can

```
SELECT count(*) FROM mimic3_demo.ADMISSIONS
```

You can also count any particular variable. For instance if we want to query the number of HADM_IDs we can query:

```
SELECT count(HADM_ID) FROM mimic3_demo.ADMISSIONS
```

You'll find that this returns the number 129 - the same as the number of rows in the ADMISSIONS table. This is because each admission gets a unique HADM_ID. You may also notice that the column name is "f0_". This is the default way that Google BigQuery returns function columns. If you had a second function - let's say find the minimum (or earliest) time of admission, you would get a second column, "f1_".

```
SELECT count(HADM_ID), min(ADMITTIME) FROM mimic3_demo.ADMISSIONS
```

Because these names are not very descriptive, it's useful to rename your columns when you apply functions.

```
SELECT count(HADM_ID) as NUMBER_OF_ADMISSIONS, min(ADMITTIME) as
EARLIEST_ADMISSION

  FROM mimic3_demo.ADMISSION
```

You can apply filters to these queries and it will only return the counts of rows that match the criteria. For example, if we want to see the number of admissions where the ADMISSION_LOCATION was the emergency room, we can run the following query:

```
SELECT count(HADM_ID) FROM mimic3_demo.ADMISSIONS

   WHERE ADMISSION_LOCATION = "EMERGENCY ROOM ADMIT"
```

This should return a count of 81.

Try it out for yourself:

1. Using the ICUSTAYS table, count the number of number of ICU Stays in the database.
2. What is the earliest INTIME from the ICUSTAYS table?
3. What is the latest OUTTIME from the ICUSTAYS table?
4. How many ICU stays started (FIRST_CAREUNIT) in the MICU?

Distinct and Grouped Values

As described above, count() returns a count of all rows that match the criteria. Sometimes it's useful to know that total number of unique values in a particular column. In this case you can use the term "distinct" to limit the count to only unique values.

Let's see how many patients are actually included in the ADMISSIONS table. We can count the number of unique or *distinct* SUBJECT_IDs.

```
SELECT count(distinct SUBJECT_ID) FROM mimic3_demo.ADMISSIONS
```

We can see that 100 patients have had the 129 admissions in the database. This means that some patients have had more than one admission. If we want to see how many admissions each patient has had, we can apply the "group by" syntax.

```
SELECT SUBJECT_ID, count(distinct HADM_ID) as ADMISSION_COUNT FROM
mimic3_demo.ADMISSIONS

  GROUP BY SUBJECT_ID
```

In this query we take each SUBJECT_ID and count the number of unique HADM_IDs that have the same SUBJECT_ID. If you have a lot of data it may be helpful to sort this information.

```
SELECT SUBJECT_ID, count(distinct HADM_ID) as ADMISSION_COUNT FROM
mimic3_demo.ADMISSIONS

 GROUP BY SUBJECT_ID

 ORDER BY ADMISSION_COUNT
```

This will list the rows by the number of admissions from smallest to largest. If you want to do the reverse order, simply apply the "desc" argument.

```
SELECT SUBJECT_ID, count(distinct HADM_ID) as ADMISSION_COUNT FROM
mimic3_demo.ADMISSIONS

 GROUP BY SUBJECT_ID

 ORDER BY ADMISSION_COUNT desc
```

Querying and Joining Multiple Tables with Google BigQuery

Joining Tables

As described in the video Joining Tables with SQL, understanding SQL joins may take a bit of time. We've created a optional reading for you to understand basic SQL syntax for joins that includes the animations from the video, along with additional examples.

This reading is to teach you how Google BigQuery handles joins. The "Standard SQL" dialect is under active development and doesn't always follow the general SQL formatting that we've covered in those resources. Specifically the issue for the Standard SQL dialect is an issue of duplicate columns.

Duplicate Columns Error

As you may have noticed in the MIMIC-III dataset, each table has the variable ROW_ID. This variable is unique to each table and should not be used to join across tables. As of early 2019, Google BigQuery Standard SQL does not handle duplicate columns across tables. As such when you try to run a join using MIMIC data where you want all columns you must specifically "exclude" the ROW_ID column. If we tried to perform an inner join of PATIENTS and ADMISSIONS, with the general SQL syntax we'd expect the query to be something like:

```
SELECT * FROM mimic3_demo.PATIENTS a

      INNER JOIN mimic3_demo.ADMISSIONS b on a.SUBJECT_ID = b.SUBJECT_ID
```

However if you try to run this query at bigquery.cloud.google.com you would likely get an error about duplicate columns ROW_ID and perhaps SUBJECT_ID. To solve the first error you can use the following query to "exclude" the ROW_ID variable.

```
SELECT * except(ROW_ID) from mimic3_demo.PATIENTS a

   INNER JOIN mimic3_demo.ADMISSIONS b on a.SUBJECT_ID = b.SUBJECT_ID
```

This query should run, however anecdotally our team has experienced that sometimes Google BigQuery still will give a duplicate column error for SUBJECT_ID. If you experience this error you can instead use the "using()" function in your join rather than the "on" statement. For instance:

```
SELECT * except(ROW_ID) FROM mimic3_demo.PATIENTS a

    INNER JOIN mimic3_demo.ADMISSIONS b using(SUBJECT_ID)
```

This query should run without problems.

Now that we have that out of the way - let's try some other examples!

Table and Column References and Aliases

Although the Standard SQL syntax has some challenges with duplicate columns syntax - one benefit of the tool is that when you select all columns in the join - the query returns all of the column names as they originally were shown in the table.

This will be true even if you select specific columns (as long as there are no duplicates). However you do still need to reference the table name (or alias) in your query. For example, let's say we want to select the SUBJECT_ID and GENDER from the PATIENTS table along with the HADM_ID, ADMITTIME and DISCHTIME from the ADMISSIONS table we would need to select these columns with the table name.

```
SELECT a.SUBJECT_ID, a.GENDER, b.HADM_ID, b.ADMITTIME, b.DISCHTIME FROM

  mimic3_demo.PATIENTS a

    INNER JOIN mimic3_demo.ADMISSIONS b using(SUBJECT_ID)
```

Although we called each column like "a.SUBJECT_ID", the table returned just calls the column SUBJECT_ID. This is actually really nice for importing data for analysis because the column names are interpretable and easy to work with.

Unfortunately it does mean that if you actually need data from duplicate columns, like ROW_ID, even explicitly calling the two columns as in the next query will also fail:

```
SELECT a. ROW_ID, b.ROW_ID, a.SUBJECT_ID, a.GENDER, b.HADM_ID, b.ADMITTIME, b

  .DISCHTIME FROM mimic3_demo.PATIENTS a

    INNER JOIN mimic3_demo.ADMISSIONS b using(SUBJECT_ID)
```

If you need both columns you can use a handy, though not intuitive or standard, shortcut. If you put the table name/s (or aliases) in your select statement, it will automatically select all columns in the table, and prepend the column name with the table name(or alias). For example if we wanted something similar to the above query we could use:

```
SELECT a, b FROM mimic3_demo.PATIENTS a

    INNER JOIN mimic3_demo.ADMISSIONS b using(SUBJECT_ID)
```

This will return all columns from both tables. Columns from PATIENTS will all start with "a.COLUMN_NAME" while columns from ADMISSIONS will all start with "b.COLUMN_NAME".

Try it out for yourself:

1. Try performing an inner join of DIAGNOSES_ICD and D_ICD_DIAGNOSES. What column/s did you use for the join?
2. Try performing an inner join of DIAGNOSES_ICD and D_ICD_DIAGNOSES that includes the values for ROW_ID from both tables.
3. How many rows in DIAGNOSES_ICD have an ICD9 code that contains the word "hypertension"?

Combining Tables with Nested SQL Statements

One other useful technique to know is that you can actually nest SQL select statements together. Let's say that we want to select all patients from the PATIENTS table that also had an entry in

INPUTEVENTS_CV. We could do an inner join between these two tables, but then we get all of the data from INPUTEVENTS_CV. Instead we can write a nested SQL statement. We'd start with the general format to select records from PATIENT when they meet a specific criteria:

```
SELECT * FROM mimic3_demo.PATIENTS where SUBJECT_ID ...
```

Then in the where statement we can actually create a second SQL statement that identifies all unique SUBJECT_IDs from the INPUTEVENTS_CV table. This nested query is:

```
SELECT SUBJECT_ID FROM mimic3_demo.INPUTEVENTS_CV GROUP BY SUBJECT_ID
```

This query returns just a list of SUBJECT_IDs.

Since this is actually a group of many possible matches, instead of using the traditional "where SUBJECT_ID = *some_value*" we actually use the syntax "in". This means that any value of SUBJECT_ID that is in the list of SUBJECT_IDs in the nested SQL statements is included. Pulling these pieces together the final nested query is:

```
SELECT * FROM mimic3_demo.PATIENTS

 WHERE SUBJECT_ID in

     (SELECT SUBJECT_ID FROM mimic3_demo.INPUTEVENTS_CV GROUP BY SUBJECT_ID)
```

Joining Tables with SQL

An important SQL skill is combining two or more tables in a single query. This process is called “joining” tables. To be able to join two tables there has to be at least one variable in common across both tables.

To make joins easier to understand, let’s use a simple example. We have two tables, X and Y. Each table has an identifier column - a number inside a colored box. Each table also has some data, shown in grey.

To combine these two tables, we need to join them on the column they have in common - the colored identifier column. What data matches depends on the type of join we want to perform. There are three main types of joins you should be familiar with: inner, left, and right joins.

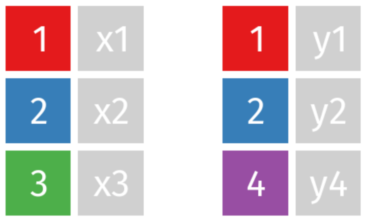
Inner Joins

Inner joins take all rows in common between the two tables and shows all of the columns from each table. The syntax for inner joins are:

```
SELECT * FROM x

    INNER JOIN y on x.ID = y.ID
```

Notice that we used the table names to tell which ID column we were referencing. You can see the effect of this join with the following animation:



Left Joins

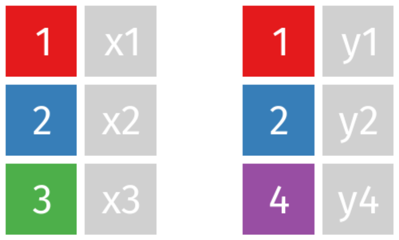
Left joins take all rows from the first "left" table (the one immediately after the "SELECT * FROM" statement) and adds data from the second "right" table (the one immediately after the "JOIN" statement) for all rows in common. If the left table does not have a match in the right table, the values for columns from the right table are left blank (NA).

Let's look at our simple example. If we do left join between x an y our code is:

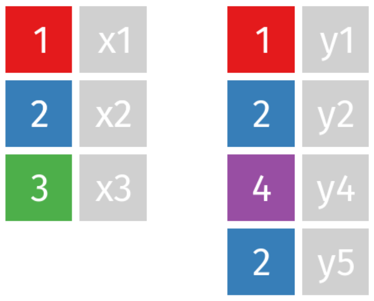
```
SELECT * FROM x

    LEFT JOIN y on x.ID = y.ID
```

And we see that table x and y have two IDs in common 1 and 2, when we do our join IDs 1 and 2 will have the data columns x and y. ID 3 does not have a match in table y, so it will only have data for column x.



A more complicated example is when there are multiple rows that have a match in one table. In the following animation table y actually has two rows with the ID=2. When this happens (regardless of the type of join) the data in the other table is duplicated and combined with both records. So in this case, x2 appears for both rows of ID=2 in the join.

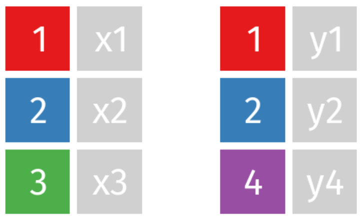


Right Joins

As you can imagine, right join simply reverses the left join. Now all rows in the second "right" table are returned, while only data that has a match from the the first "left" table is returned. The query for a right join is:

```
SELCT * FROM x


    RIGHT JOIN y on x.ID=y.ID
```



References and Aliases

Table Name References

Finally, you should know an important nuance to performing queries with joins. You can apply all the same column and row selection approaches used when querying a single table, but when your query has a join, you have to explicitly label which table holds the particular column used. To do this, any place where you would have had the column name, you put TableName-dot-ColumnName - just like we did in the example with x.ID = y.ID

Let's look at a more realistic example. Consider you have two tables "Patient" and "Encounter".

Patient

Patient_ID	First_Name	Last_Name
1	Bob	Smith
2	Sally	Williams
3	Jessie	Jones

Encounter

Encounter_ID	**Patient_ID**	**Visit_Date**
1	1	05/18/2018
2	1	08/11/2018
3	1	09/16/2018
4	2	10/11/2018

If you want to only select specific columns during your join or "group by" or "order by" a particular column, you must reference the table the variable is from. Let's say that we only wanted the Patient_ID, Last_Name, and Visit_Date. We would run the following query:

```
SELECT Patient.Patient_ID, Patient.Last_Name, Encounter.Visit_Date FROM Patient

    INNER JOIN Encounter on Patient.Patient_ID = Encounter.Patient_ID
```

And would get the following result:

Patient.Patient_ID	**Patient.Last_Name**	**Encounter.Visit_Date**
1	Smith	05/18/2018
1	Smith	08/11/2018
1	Smith	09/16/2018
2	Williams	10/11/2018

As you can imagine this gets tedious quickly when you have more complex queries. See the next section on Table Name aliases for a short cut to make this process easier!

Table Name Aliases

SQL has a process for creating a table alias - or a shortcut - to refer to the table. Simply put a space after the table name, followed by your desired alias and it will know when you use the alias in the query.

For example with the query from before, we can use the alias function to refer to Patient as p and Encounter as e:

```
SELECT p.Patient_ID, p.Last_Name, e.Visit_Date from Patient p

  INNER JOIN Encounter e on p.Patient_ID = e.Patient_ID
```

Which will give the result:

p.Patient_ID	p.Last_Name	e.Visit_Date
1	Smith	05/18/2018
1	Smith	08/11/2018
1	Smith	09/16/2018
2	Williams	10/11/2018

Column Name Aliases

Depending on the type of SQL you are running, you may notice that when performing a join SQL automatically puts the table name or table alias before the variable name. You can actually change the name of any column with the "as" command. When you are selecting the column you can follow the column name for instance "Patient.Last_Name as LastName" will rename the column in your result. Putting together the full query:

```
SELECT p.Patient_ID as Patient_ID, p.Last_Name as Last_Name, e.Visit_Date as

  Visit_Date from Patient p

  INNER JOIN Encounter e on p.Patient_ID = e.Patient_ID
```

This should produce the following result:

Patient_ID	Last_Name	Visit_Date
1	Smith	05/18/2018
1	Smith	08/11/2018
1	Smith	09/16/2018
2	Williams	10/11/2018

Acknowledgments

SQL joins are a challenging concept to understand and to teach. I want to thank Garrick Aden-Buie for creating all of the .gifs we used in this course and making them freely available in the TidyExplain package. Make sure to check out his website and GitHubto see some of the other cool projects he's working on!