SQL FOR DATA SCIENCE

Matt Richards

July 20, 2020

TITLE: SQL for Data Science
AUTHOR: Matt Richards (McMaster University)
NUMBER OF PAGES: 86

Abstract

Acknowledgements

Contents

Chapter 1

Basics of SQL

1.1 Week 1

1.1.1 What is SQL?

SQL stands for Structured Query Language. It is a method for communicating between the user and the data base. Many of the commands are fairly easy to interpret but it is a non-procedural language. Meaning you can't write complete applications with it.

SQL is all about data. It's main focus is to read, write, retrieve, or update data from a data base. Many positions have a strong dependence on understanding SQL. Database Administrators are responsible for permissions and governance of databases while data scientists are mainly the end users of a database.

SQL is very fundamental in data science as it provides the means towards communicating with databases. The main thing they are using SQL for is data retrieval.

Database Management Systems (DBMS) each have their own syntax. These are the platforms in which SQL works. For this course I will be using SQLite. Some of these are: MySQL, SQLite, Microsoft SQL Server, ect...

1.1.2 Data Models

This is a very fundamental lesson for understanding SQL queries. The key idea is to always "Think before you do". Always ask yourself "What is the problem you are trying to solve?". Make sure you understand the type of data you are trying to get and what are the underlying business processes for this? By going through this process, you will be able to construct queries much quicker as you can focus on the task that needs to be done.

Databases can be thought of like file cabinets which keep many different types of data structures within them. Data models organize and structure information into multiple related tables and they should always represent a real world problem as close as possible. Es-

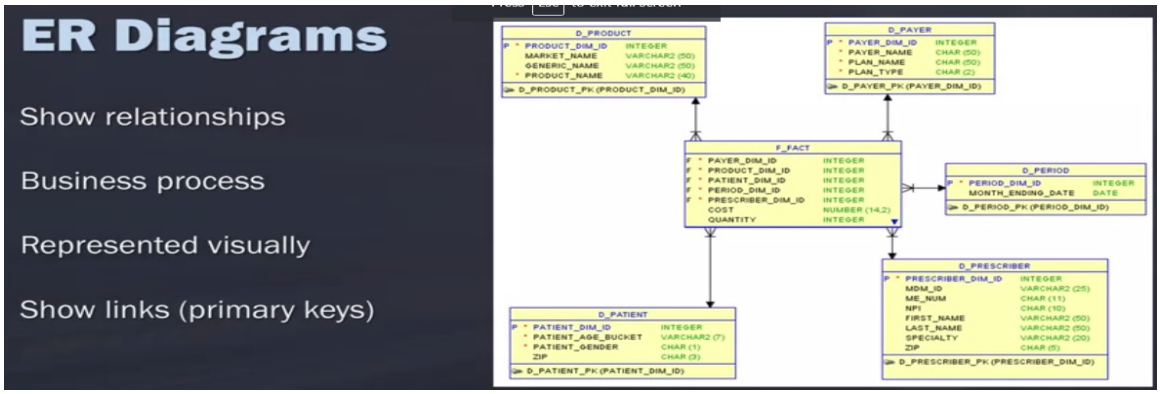


Figure 1.1

sentially, data models are the way in which we interlink different databases for the purposes of modeling different systems.

NoSQL is one the most modern forms of data modeling and it is what we call a relational database model. These relational database models allow you to easily write, query, retrieve, and update data from a database. NoSQL stands for Not Only SQL, which provides a mechanism for retrieval and storage of unstructured data through other means than tables.

Transactional database models are called operational in that data might not be easily readable or easy to extract from it. An example of this would be insurance claims in healthcare.

1.1.3 Relational Database Models

These are the main database models that we will be working with here. There are three key building blocks:

- Entity: Noun or events which are distinguishable, unique and distinct.
- Attribute: A characteristic of an entity.
- Relationship: Describes the association among entities in the form of One-to-many, Many-to-many, or One-to-one.

These relationship definitions are self explanatory.

Relationships are often described through ER diagrams as shown in Figure 1.1. Being able to look at a diagram like this and determine the relationships is very important. The types of notations are presented in Figure 1.2.

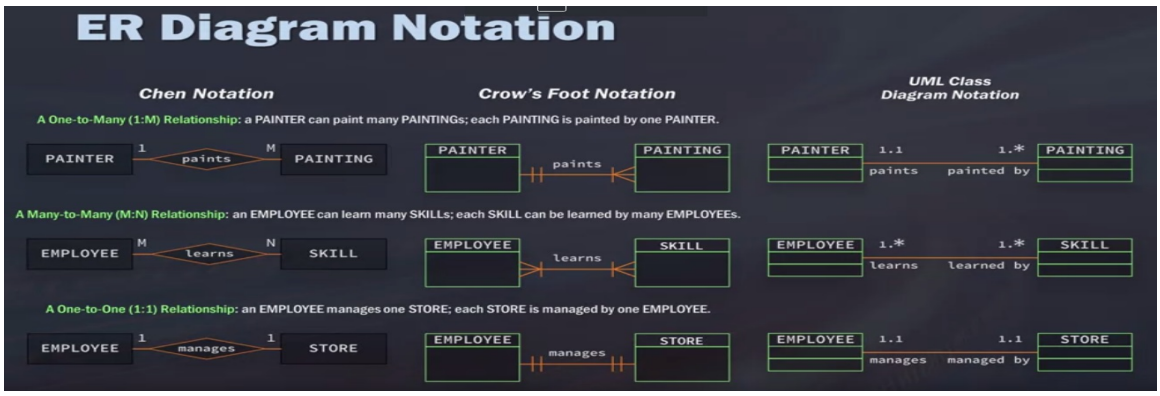


Figure 1.2

1.1.4 Retrieving data

The majority of what data scientists are doing is retrieving data. The Select statement needs two pieces of information, what you want and where you want to select it from.

Listing 1.1: Basic retrieval

```
SELECT prod_name
FROM Products;

/*This will output all of the columns with the name "prod_name" from Products*/

SELECT prod_name,prod_id,prod_price
FROM Products;

/*This will output all of the columns with the names listed after select from Pr

SELECT *
FROM Products;

/*This will request all columns from Products */

SELECT prod_name
FROM Products
LIMIT 5;

/* This will only output the first 5 rows of prod_name from Products */
```

1.1.5 Creating Tables

It is very important to make tables within a SQL environment. This is the main way to store information within data models.

Listing 1.2: Basic Table construction

```
CREATE TABLE Shoes
(
ID        char(10)          Primary Key,
Brand     char(10)          NOT NULL,
Type      char(250)         NOT NULL,
Color     char(250)         NOT NULL,
Price     decimal(8,2)      NOT NULL,
Desc      Varchar(750)      NULL
);
/*Every column is either NULL or NOT NULL. This determines whether the values
/*in the columns can be NULL or not.*/
/*Primary keys can't accept NULL values.*/

INSERT INTO Shoes
VALUES ('14525974'
        ,'Gucci'
        ,'Slippurs'
        ,'Pink'
        ,'695.00'
        ,NULL
            );
/* You can insert indiviual items into a table.*/
/*However, there is no guarentee, were it will end up*/
INSERT INTO Shoes
        (ID
        ,Brand
        ,Type
        ,Color
        ,Price
        ,Desc
        )
VALUES ('14525974'
        ,'Gucci'
```

```
        ,'Slippurs'
        ,'Pink'
        ,'695.00'
        ,NULL
            );
/*This is a much better way to do that since you are directly saying*/
/* Where each value should go*/
```

1.1.6 Creating Temporary Tables

Temporary tables will be deleted when the current session is terminated. These are good to use because they are much faster than creating a real table. It can be quite good since they are useful for writing complex queries using subsets and joins.

Listing 1.3: Basic temporary table construction

```
CREATE TEMPORARY TABLE Sandals AS
 (
    SELECT *
    FROM Shoes
    WHERE shoe_type = 'sandals'
)

/* This will take all the columns from Shoes and output a temporary table */
-- Which have a shoe_type of 'sandals'
/* -- represents comment for a single line.
```

1.2 Week 2

1.2.1 Basics of Filtering

Many times, datasets are filled with millions if not billions of columns and records. We want to simplify our searches through filtering. This allows us to optimize the type of data we are working with and provides a means t doing his securely within the data structures framework. Instead of just going and grabbing a whole table and pulling every column and row from it, we can get really specific about what data we want to obtain from that table. We do this with the WHERE clause.

Listing 1.4: Basic Filtering with WHERE

```
SELECT ProductName, UnitPrice, SupplierID
FROM Products
WHERE ProductName = 'Tofu';

-- Filtering on a single condition.

SELECT ProductName, UnitPrice, SupplierID
FROM Products
WHERE UnitPrice >=75;

-- Filtering on a single value.

SELECT ProductName, UnitPrice, SupplierID
FROM Products
WHERE ProductName <>'Alice Mutton';

-- This filter pulls all records except for ones where ProductName is Alice Mutto

SELECT ProductName, UnitPrice, SupplierID
FROM Products
WHERE UnitsInStock BETWEEN 15 AND 80;

-- This filter provides a range for UnitsInStock between 15 and 80.

SELECT ProductName, UnitPrice, SupplierID
FROM Products
WHERE ProductName IS NULL;

-- Filters all NULL values for ProductName
```

1.2.2 Advanced Filtering

Listing 1.5: Advanced Filtering with IN, OR, and NOT

```
SELECT ProductName, UnitPrice, SupplierID
FROM Products
WHERE ProductName IN (9,10,11);
```

```
-- We limit our results to the suppliers that are 9,10,11

SELECT ProductName, ProductID, UnitPrice, SupplierID
FROM Products
WHERE ProductName ='Tofu' OR 'Konbu';

-- Here we are selecting each term from Tofu or Konbu.
-- A DBMS will not evaluate the second condition in a WHERE clause if the first
-- condition is met.
-- In executes faster than OR.

SELECT ProductID, UnitPrice, SupplierID
FROM Products
WHERE SupplierID = 9 OR SupplierID = 11
AND UnitPrice > 15;

-- This will give some UnitPrices that are not greater than 15.
-- The reason why this is happening is because SQL is executing the OR before the

SELECT ProductID, UnitPrice, SupplierID
FROM Products
WHERE (SupplierID = 9 OR SupplierID = 11)
AND UnitPrice > 15;

-- The parenthesises indicate the order of operations.
-- This will fix the problem.

SELECT *
FROM Employees
WHERE NOT City='London' AND NOT City='Seattle';

--This will give every query that does not have a city
--in London or Seattle.
```

1.2.3 Wildcards in SQL

This will be useful in filtering data where you know a bit of information about the data like its first and last letters, or if it is similar to something else. These Wildcards are very useful

when retrieving data. Generally, wildcards take longer to run and it is often better to use other operations if you can.

Listing 1.6: Using Wildcards when filtering

```
--These wildcards can only be used with strings.

'%Pizza'
--Grabs anything ending with the word pizza.
'Pizza%'
--Grabs anything after the word pizza.
'%Pizza%'
--Grabs anything before and after the word pizza.

'S%E'
--Grabs anything that starts with "S" and ends with "E" (like Sadie)
't%@gmail.com'
--Grabs gmail addresses that start with t and end with @gmail.com.

--Wildcards won't grab NULL values.

WHERE size LIKE '_pizza'
Output:
    spizza
    mpizza

-- This is another way of writing wildcards.
--Square brackets work as well (SQLit doesn't support this)
--It is all dependent on the database management systems.
```

1.2.4 Sorting with ORDER BY

The ORDER BY clause will allow you to sort data for your analysis purposes. Many times, data isn't returned any logical order, so it is always good to be specific about how you want it ordered. The ORDER BY rule takes the name of one or more columns, make sure you add a comma after each column and that it is the last clause in a select statement.

Listing 1.7: Using ORDER BY for sorting

```
ORDER BY 2,3
```

```
--Here we are sorting it by the 2nd column and 3rd column.
--You can also use the names specifically.
--DESC=descending order and ASC= ascending order.
--Only applies to the column names that directly preced it.
```

1.2.5 Basic Math Operations

Listing 1.8: Math Operations

```
SELECT ProductID, UnitsOnOrder, UnitsPrice,
UnitsOnOrder * UnitPrice AS Total_Order_Cost
FROM Products;

--Here we are creating a new column called Total_Order_Cost which is the product
--of UnitsOnOrder and UnitPrice.
--The AS command determines what we want to call this new value as.

SELECT ProductID, UnitsOnOrder, UnitsPrice,
(UnitPrice - Discount)/Quantity AS Total_Cost
FROM Products;
```

1.2.6 Agrregate Functions

These aggregate functions can be used for all sorts of things and build off of math operators. They are just pre-built functions to SQL. NULL values are ignored by many of these aggregate functions.

Listing 1.9: Aggregate functions

```
AVG()
--Averages a column of values.
COUNT()
--Counts the number of values.
MIN()
--Finds the minimum value.
MAX()
--Finds the max value.
SUM()
```

```
--Sums the column values.

SELECT AVG(UnitPrice) AS avg_price
FROM products;

--This takes the average of the UnitPrice

SELECT COUNT(*) AS total_customers
FROM Customers;

--Counts all the rows in a table containing values or NULL values.

SELECT COUNT(CustomerID) AS total_customers
FROM Customers;

--Counts all the rows in a table containing values ignoring NULL values.

SELECT MAX(UnitPrice) AS max_prod_price
,Min(UnitPrice) AS min_prod_price
FROM Products;

--Returns min and max value as their names.

SELECT SUM(UnitPrice*UnitsInStock) AS total_price
FROM Products
WHERE SupplierID=23;

--This will return the total sum from the individual products of
--UnitPrice and UnitsInStock and calling it total price
--under the condition SupplierID=23.

SELECT COUNT(DISTINCT CustomerID)
from Customers;

--If DISTINCT is not specified, SQL will assume you want all the data.
--ex: If you want to count customers, you need to specify distinct
--Otherwise you may overcount one or more customers
--Can't use DISTINCT on COUNT(*).
```

1.2.7 Grouping Data with SQL

Many times we want to apply operations to the data within multiple tables. We can perform many aggregate functions using he GROUP BY clause.

Listing 1.10: Grouping data

```
SELECT Region
,COUNT(CustomerID) AS total_customers
FROM Customers
GROUP BY Region;
--If we were to just have the SELECT Region without the GROUP BY region
--we would have an error because we didn't specify ow we want to count the Custon
--GROUP BY clauses can contain multiple columns.
--Every column in your SELECT statement must be present in a GROUP BY clause.
--WHERE does not work for groups because it applies to rows.

SELECT CustomerID
,COUNT(*) AS orders
FROM Orders
GROUP BY CustomerID
HAVING COUNT(*) >=2;

--Where filters before data is grouped.
--Having filters after data is grouped.

SELECT SupplierID
,COUNT(*) AS Num_prod
FROM Products
WHERE UnitPrice >= 4
GROUP BY SupplierID
HAVING COUNT(*) >=2;

--This will select the SupplierID's and the count of all columns as Num_prod
--given the UnitPrice >=4. It will group these values by the SupplierID
--which have a count greater than 2.
```

1.2.8 Summary

If we are using an aggregate function, we want to make sure we have that GROUP BY clause.

1.3 Week 3

In this lesson, we will learn how to make a subquery, merge data from tables with the join statement. We are going to learn how to use multiple tables and this is were the value in relational databases comes from.

1.3.1 Subqueries

Subqueries are queries which are embedded into other queries. Data Scientists often use subqeuries in order to select specific columns for certain criteria. They are often used for adding additional criteria which may not be in a given table.

Listing 1.11: Subqueries

```
--We can see how useful subqueries are by considering
--a case where we need two tables.
```

Key SQL Clauses

Clause	Description	Required
SELECT	Columns or expressions to be returned	Yes
FROM	Table from which to retrieve data	Only if selecting data from a table
WHERE	Row-level filtering	No
GROUP BY	Group specification	Only if calculating aggregates by group
HAVING	Group-level filter	No
ORDER BY	Output sort order	No

Figure 1.3

```
------------
--Table 1--
------------
SELECT DISTINCT customerID
FROM Orders
WHERE Freight >100;

--Need to know region of each customer with an order of freight >100.

------------
--Table 2--
------------
SELECT CustomerID,CompanyName,Region
FROM Customers
WHERE CustomerID IN ('RICSU','ERNISH',--...);

--We need to manually copy and paste in the results of Table 1 into Table 2.
--This is where a subquery comes in, it reduces the tediousness.

------------
--Subquery--
------------

SELECT CustomerID,CompanyName,Region--Next take the outer query given the conditi
--of the inner query.
--This is done through the WHERE clause.
FROM Customers
WHERE customerID in (SELECT customerID --Start with inner most query
    FROM Orders
    WHERE Freight > 100);

--We combine all three queries to make the code more efficient.
--Always perform the innermost SELECT portion first.
--This DBMS is performing two operations:
    --1. Getting the order numbers for the product selected.
    --2. Adding that to the WHERE clause and processing the overall SELECT statem
```

1.3.2 Subquery Best practices

We can apply subqeuries recursively for calculations if need be and what the best practices are. There is no limit to the number of subqueries you can have. Performance slows quite drastically when these queries are nested deeply.

IMPORTANT: Subquery selects can only retrieve a single column.

Listing 1.12: Subquery practices

```
SELECT Customer_name, Customer_contact
FROM Customers
WHERE cust_id IN
    SELECT customer_id
    FROM Orders
    WHERE order_number IN(SELECT order_number
    FROM OrderItems
        WHERE prod_name= 'Toothbrush');

--Start with the innermost subquery when interpreting code.
--1. get order_number from OrderItems given prod_name=toothbrush.
--2. get the customer_id from Orders given order_number statisfies condition 1.
--3. print out Customer_name and Customer_contact with conditions 1. and 2.

SELECT COUNT(*) AS orders
FROM Orders
WHERE customer_id = '143569';

--Subqueries can be used as calculations.
--We could count the number for a particular customer_id.
--We still wouldn't have the customer name and state.

SELECT customer_name,customer_state
    (SELECT COUNT (*) AS orders
    FROM Orders
    WHERE Orders.customer_id=Customer.customer_id) AS orders
FROM customers
ORDER BY Customer_name;

--We want to select the customers name and state,
```

```
--along with the subquery itself as orders.
--The subquery takes the count given the Orders customer_id is the Customers cust
--The first two columns are from customers, and the third column is from Orders.
--Lastly, we order this information by Customer_name.

--Subqueries are very powerful but can be computational extensive.
```

1.3.3 Joining Tables

Understanding the database is very important when writing queries. Most of the time, the data we are after is stored in multiple tables. This provides a more efficient manner for understanding where the data comes from, its nature, how its stored, and the best way to manipulate it.

How we break down the data into tables is often modeled through a business process. It provides an easy way to manipulate the data, scalability with the information, and efficient storage as well. Often times we will take the tables and reduce them in a way it has its own process. Unique keys can be used to distinguish these data records and it is critical when joining many tables together.

Join will allow you to retrieve data from multiple tables in one query. It associates the correct records from each table on the fly. It is important to note that Joins do not create anything permanent and they only persist for the duration of the query.

Listing 1.13: Join clauses

```
--Here we will cover the different types of Joins.

-------------------------
-- Cartesian Join --
-------------------------

--Takes x rows from the first table and matches it with y rows from the second ta
--and makes x.y rows.
--This Join clause is computationally taxing and is not often used.

SELECT product_name, unit_price, company_name
FROM suppliers CROSS JOIN products;

--This is taking every column selected in ''suppliers" and multiplying it
--with every column in ''products".
```

```
--It will spit out nonsense since it multiplies each of these values.

-------------------
-- Inner Join --
-------------------

--Inner joins are one of the most frequently used joins clauses.
--An inner join is used to select records that have matching values in two tables
--Its going to look for the key to identify if the records are matching.

SELECT suppliers.CompanyName,ProductName,UnitPrice
FROM Suppliers INNER JOIN Products ON Suppliers.supplierid = Products.supplierid;

--The key used here is supplierid as its taken from each table.
--INNER JOIN is in the FROM clause and it uses the ON clause.
--You can apply this recursively but it requires specific keys.

--INNER JOIN with multiple tables.

SELECT o.OrderID, c.CompanyName, e.LastName
FROM ((Orders o INNER JOIN Customers c ON o.CustomerID = c.CustomerID)
INNER JOIN Employees e ON o.EmployeeID = e.EmployeeID);

--We are labeling each table by o, c, and e.
--Make sure you think about what you are doing when you join tables.
```

1.3.4 Aliasing and Self Joining

Aliases can help by shortening names and by prequalifying the data. It is not rewriting the label of that data since it only works within the query.

Listing 1.14: Aliasing tables and Self Joins

```
--Consider the following query.

SELECT vendor_name, product_name, product_price
FROM Vendors, Products
WHERE Vendors.vendor_id = Products.vendor_id;
```

```
--We can simplify the query by declaring the table as an alias as

SELECT vendor_name, product_name, product_price
FROM Vendors AS v, Products AS p
WHERE v.vendor_id = p.vendor_id;

--We declare this in the FROM clause.

-------------------
-- Self Joins --
-------------------

--We can Join a table to itself through a SELF JOIN.
--We take the table and treat it like two separate tables.

SELECT column_name(s)
FROM table1 T1, table2 T2
WHERE condition;

--Example

SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.city
FROM Customers A, Customers B
WHERE A.CustomerID = B.CustomerID
AND A.City = B.city
ORDER BY A.City;

--This query matches customers that are from the same city.
--This is a great tool when you want to match certain elements with itself.
```

1.3.5 Advanced Joins

Listing 1.15: Advanced Joins, Left, Right, and Full Outer Joins

```
--SQL Lite only does a Left Join.

-------------------
-- Left Join --
```

```
-----------------

--Left Joins return all records from the left table that are matched
--with records from the right table.
--The result is NULL from the right side if there is no match.

--The difference between this and the INNER JOIN is that I will take all the reco
--selected from the left tables regardless if it matches the right table.

--This statement will select all customers, and any orders they might have.
SELECT C.CustomerName, O.OrderID
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID=0.CustomerID
ORDER BY C.CustomerName;

-----------------
--Right Join--
-----------------

--Not supported in SQL Lite.

--A Right Join does the same thing except vice versa.

--This will return all employees and any orders they might have placed.
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName
FROM Orders
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
ORDER BY Orders.OrderID;

--Note: You could use a left join simply by switching the Employees and Orders.

---------------------
--Full Outer Join--
---------------------

--Not supported in SQL Lite.

--A Full Outer Join returns all records when there is a match
```

```
--in either left or right table records i.e. ''Give me everything".

SELECT Customers.CustomerName, Orders.OrderID
FROM Customers
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerName;
```

1.3.6 Unions

A UNION is used to combine the result of two or more queries into one table and one statement. Each SELECT statement within UNION must have the same number of columns. The data structures have to be the same as well otherwise the result will be nonsensical.

Listing 1.16: Unions

```
_________
--UNION--
_________

-- You want to make sure the columns for each data are in the same order.
--General syntax

SELECT column_name(s)
FROM table1
UNION
SELECT column_name(s)
FROM table2;

--Example

SELECT City, Country
FROM Customers
WHERE Country='Germany'
UNION
SELECT City, Country
FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
--Now we are able to list all the customers and suppliers that are in Germany.
```

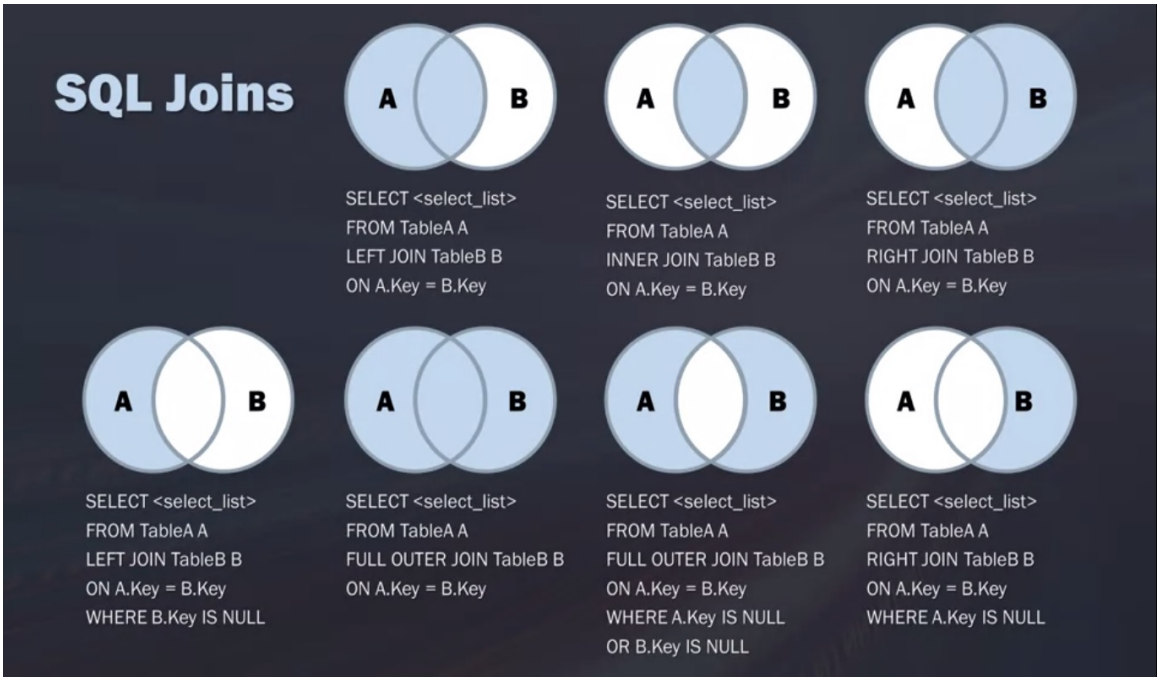


Figure 1.4

1.3.7 Join Summary

In SQL, it is very easy to get results, but it is difficult to make sure you are getting the correct results. This is where the JOIN clauses become useful. They can allow us to simplify multiple data sets and interpret data in a more logical manner. The different types of JOIN clauses are summarized in Fig. 1.4.

1.4 Week 4

Here we are going to learn about the finer details of SQL. Using Case statements (if, then, else, statements in other languages) and data governance.

1.4.1 Working With Test Strings

We are going to learn how to concatenate or combine text strings, trim text strings, use the substring function, and change the case of strings. Strings show up everywhere within data that we work with. Some of the different string functions are: concatenate, substring, trim, upper, and lower.

Listing 1.17: Working with text strings

```
-----------------------
-- Concatenation --
-----------------------

--Links together different strings together.

SELECT CompanyName, ContactName
, CompanyName || ' ( ' || ContactName || ' ) '
FROM customers

--Here we are concatenating the Companyname with the contact name.
--Output is "CompanyName"("ContactName").

-----------
-- Trim --
-----------

--Trims the leading or trailing space from a string.

SELECT TRIM("    You the best.    ") AS TrimmedString

--This is an easy way to clean data.

-----------------
-- Substring --
-----------------

--Returns the specified number of characters from a particular position of a give

SUBSTR(string name, string position, number of characters to be returned);

SELECT first_name, SUBSTR(first_name,3,4)
FROM employees
WHERE deartment_id=60;

--If the first_name is Nancy, you will get ncy.
--We indicate that we start at the third letter and pull the next four letters a
```

```
-------------------------
-- UPPER and LOWER --
-------------------------

SELECT UPPER(column_name) FROM table_name;
SELECT LOWER(column_name) FROM table_name;
SELECT UCASE(column_name) FROM table_name;

--This returns the columns with each strings letter upper cased or lower cased re
--UCASE does the same thing as UPPER.
```

1.4.2 Working with Date and Time Strings

We will learn how to describe the complexities of adjusting date and time strings, the different formats for dates and times, and use the 5 different functions to manipulate dates and times.

In general, if the data contains a date portion, the queries will work as expected. When time is introduced, it gets more complicated. There are many different ways to represent time and date.

Wednesday, September 17th, 2008

$9/17/20085:14:56P.M.EST$

$9/17/200819:14:56GMT$

2612008 (Julian format)

The most difficult part when working wih dates is to be sure that the format of the date you are trying to insert, matches the format of the date column in the database. Dates are stored based on the DBMS (Database Management System) and they are stored as various date types.

- DATE

  Format: YYYY-MM-DD

- DATETIME

  Format: YYYY-MM-DD HH:MI:SS

- TIMESTAMP

  Format: YYYY-MM-DD HH:MI:SS

Listing 1.18: SQLite Date Time Functions through examples

```
-------------------------------
-- Date Time functions --
-------------------------------

DATE(timestring, modifier, modifier, ...)
TIME(timestring, modifier, modifier, ...)
DATETIME(timestring, modifier, modifier, ...)
JULIANDAY(timestring, modifier, modifier, ...)
STRFTIME(format, timestring, modifier, modifier, ...)

--These functions will pull out the date or time from a query.

-----------------
-- STRFTIME --
-----------------

SELECT Birthdate
,STRFTIME('%Y', Birthdate) AS Year
,STRFTIME('%m', Birthdate) AS Month
,STRFTIME('%d', Birthdate) AS Day
FROM employees;

--STRFTIME function is used to extract certain parts of a date and time string.

SELECT DATE('now')
--This will give the current date for comparison reasons.

SELECT STRFTIME('%Y %m %d', 'now')
--Extract the year, month, and day from now.

SELECT STRFTIME('%H %M %S %s', 'now')
--Extract the exact time up to miliseconds.

--EX:

SELECT Birthdate
```

```
,STRFTIME('%Y', Birthdate) AS Year
,STRFTIME('%m', Birthdate) AS Month
,STRFTIME('%d', Birthdate) AS Day
,DATE(('now')-Birthdate) AS Age
FROM employees;

--In the last column, we have a new column for the age.
```

1.4.3 Case Statements

We do this often when we are trying to recode the data for specific analysis. Taking an individual variable out of its own category. This lesson will focus on CASE statements.

Listing 1.19: Case Statements

```
-----------------------
--Case Statements--
-----------------------

--General format--

CASE
WHEN C1 THEN E1
WHEN C2 THEN E2
...
ELSE [result else]
END

--Example--

SELECT employeeid, firstname, lastname, city
,CASE City
    WHEN 'Calgary' THEN 'Calgary'
ELSE 'Other'
    END calgary
FROM Employees
ORDER BY LastName, FirstName;

--This will create a new column called calgary which writes everything
```

```
--as 'Calgary' or 'Other'.
--You can use this to convert data in a one-hot representation for example.

--Example 2--

SELECT trackid, name, bytes
,CASE
WHEN bytes < 300000 THEN 'small'
WHEN bytes >= 300001 AND bytes <=500000 THEN 'medium'
WHEN bytes >= 5000001 THEN 'large
ELSE 'Other'
END bytescategory
FROM tracks;

--We can bin data in any manner that we desire.
```

1.4.4 Views

Here we will discuss how and when to use views with queries, explain how to use the AS function with views, and explain the benefits and limitations of VIEWS. Instead of creating a whole new variable, we can create a temporary table with VIEW. So we are storing the data temporarily instead of permanently.

Listing 1.20: View Statements

```
--General Format--

CREATE [TEMP] VIEW [IF NOT EXISTS] view_name(column-name-list)
AS
select-statement;

--Example--

CREATE VIEW my_view
AS
SELECT
r.regiondescription
,t.territorydescription
,e.Lastname
```

```
,e.Firstname
,e.Hiredate
,e.Reportsto
FROM Region r
INNER JOIN Territories t on r.regionid = t.regionid
INNER JOIN Employeeterritories et on t.TerritoryID = et.TerritoryID
INNER JOIN Employees e on et.employeeid = e.EmployeeID

--Here we are creating a VIEW command for each chosen column from each table.
--To actually view the data, we need to include a SELECT statement.

SELECT *
FROM my_view
DROP VIEW my_view;

--This will execute the VIEW class that we generated.
--We need to deallocate the view statement after we are done with it.

--These VIEWS don't need an ETL (Extract, Transform, Load) to execute.
--They are useful when joining tables and understanding
--what the data is and what we are trying to do.
--VIEWS can create a stepping stone in multilevel queries.

SELECT COUNT(territorydescription)
,Lastname
,Firstname
FROM my_view
GROUP BY Lastname, Firstname;

--VIEWS can help simplfy queries, are simpler than creating a new table,
--can add or remove columns without changing the schema,
--and can get around the DBMS limitations.
```

1.4.5 Data Governance and Profiling

This course focuses on the application of SQL for data science. Here we will define data governance and profiling, explain the importance of this, and the different methods for profiling data.

Data Profiling

Data Profiling is understanding the descriptive statistics or object data information when examining data for completeness and accuracy. It is important to understand the data we are working with, before we start to query it. The steps for Object Data Profiling are

- 1. Check the number of rows we are working with.
- 2. Understand the table size.
- 3. Check when the object was last updated.

We can also apply a Column Data Profile in the form of

- Column data type
- Number of distinct values
- Number of rows with NULL values
- Descriptive statistics: maximum, average, and standard deviation for column values.

Data Governance

The data governance that is used will depend on the company you are working for. This will determine your read and write capabilities, how you can clean the environments, and how the hierarchy in seniority works. These governance policies may be the reason why a query doesn't work since you are not given the permissions you need or are allowed to have.

1.5 Using SQL for Data Science

We have gained an assorted toolbox for using SQL. The difficulty comes from understanding how all of the tools we discussed will come together. When we are applying this knowledge to data science, the most important thing is fundamentally understanding the data we are working with.

1.5.1 Understanding your Data

Many times, we will need to work through a problem from beginning to end. The most important thing to start with is understanding the data we are working with. We need to know when data joins are being used, and how to differentiate integers from strings. If we invest the time in understanding the subject and where the data comes from, it will greatly help during the data analysis.

Examples of things to ask are:

- What are the data types used in each column?
- Are there any NULL values we need to deal with?
- Is the data in a date and time format.

1.5.2 Business Understanding

Business understanding is an important area as well. This is a way of saying, how does the data relate to each other and what are the fundamental process defining where it comes from. We need to ask questions about the types of business problems that we are trying to solve. How hard is it to separate the data and the business understanding? Many times, data understanding and business understanding are interchangeable.

Many times, there will be unspoken needs that we need to address. A particularly harmless question might be "We want to predict whether or not a customer is likely to buy our product." Some things we need to address are:

- Which customers?
- What Product?
- What is/should be excluded?
- What is/should be counted from past?

Going back and forth between understanding the data, and understanding the business process is considered as the most important aspect of a data scientist's job. Inherently, this will make query writing much easier.

1.5.3 Tips for SQL Code

In order to really understand a problem, we need to map out the exact data elements that we need. Where do we start with the data and our query. It is helpful to draw a diagram illustrating where each data element we need is coming from and the mappings between the courses. From here, we can determine what sorts of calculations we need.

The great thing about SQL is that it is very consistent. When writing a query, we will always start with some sort of SELECT statement and a FROM statement. We can add in special formats as we go along. If there are many subqueries or JOINS being used, we want to start with the innermost query and work outwards.

As we are working through the query, it is useful to test our queries as we are working on the code. Do we get the results we expect? Start off smaller and then go step-by-step to troubleshoot. Proper formatting, indention, and commenting makes the code easy to read

and understand for yourself and everyone else. It is useful to review the queries we write and the results being output with each execution.

Remember: The first step in data science is being able to retrieve the data.

Chapter 2

Data Anaylsis with SQL

2.1 Week 1

2.1.1 Introduction to the Course Dataset

This data is meant to mimic an online eCommerce site. It could also be adapted for sites where you save items, e.g. Pinterest. If we want to record specific events, we need different parameters that can describe these events. This is categorized in the event table. I have simply copied all of the content out of each table that we will be starting with and presented it in the Figures below.

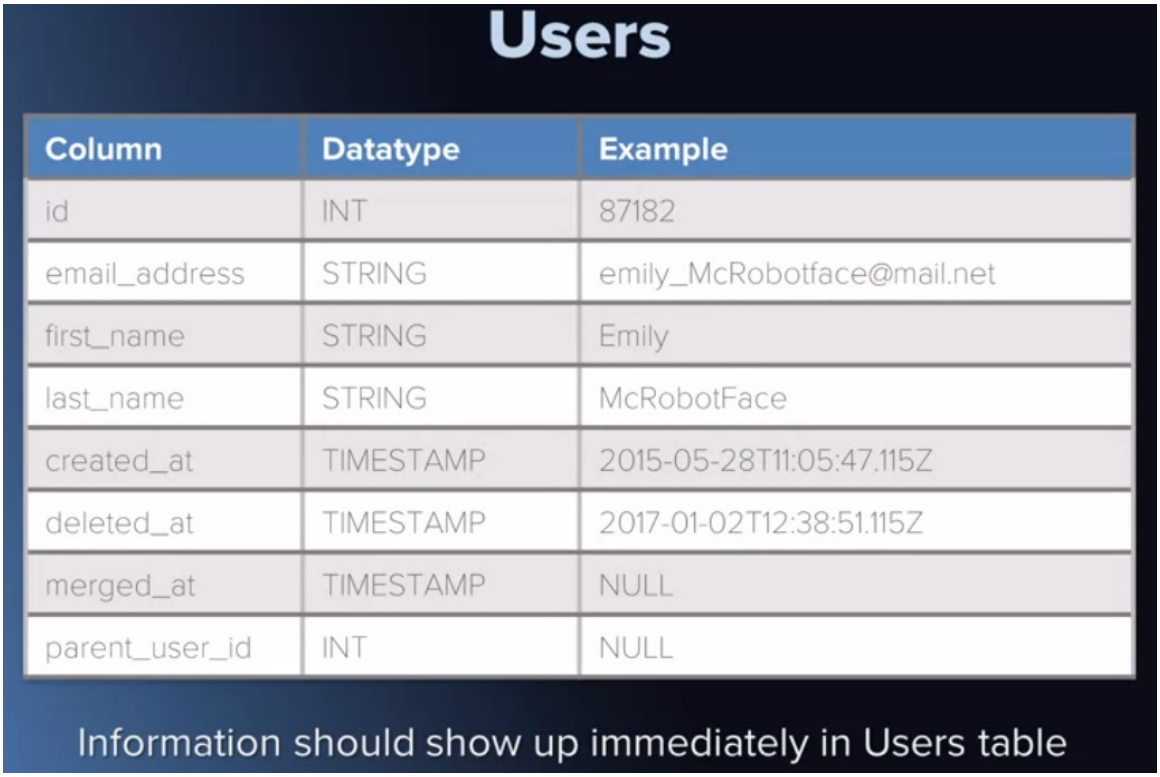


Figure 2.1: The id column in this table is often called the user ID.

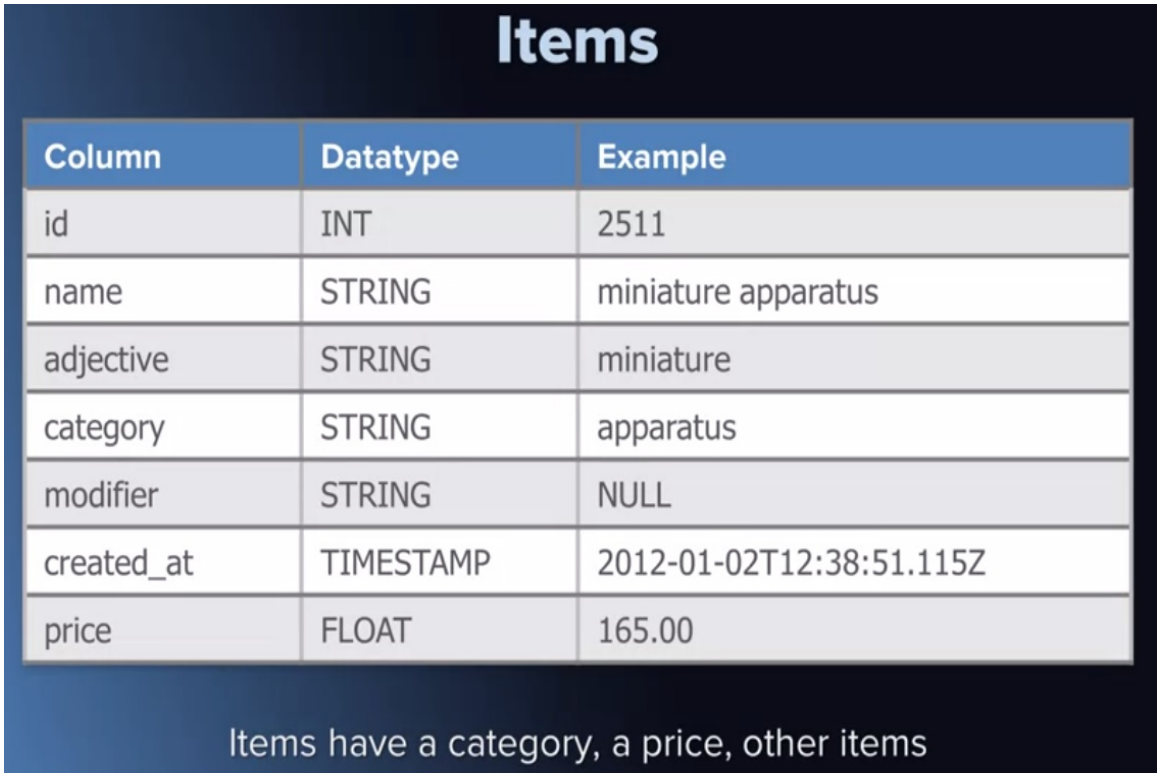


Figure 2.2: The price is at the current time stamp but is variable to change.

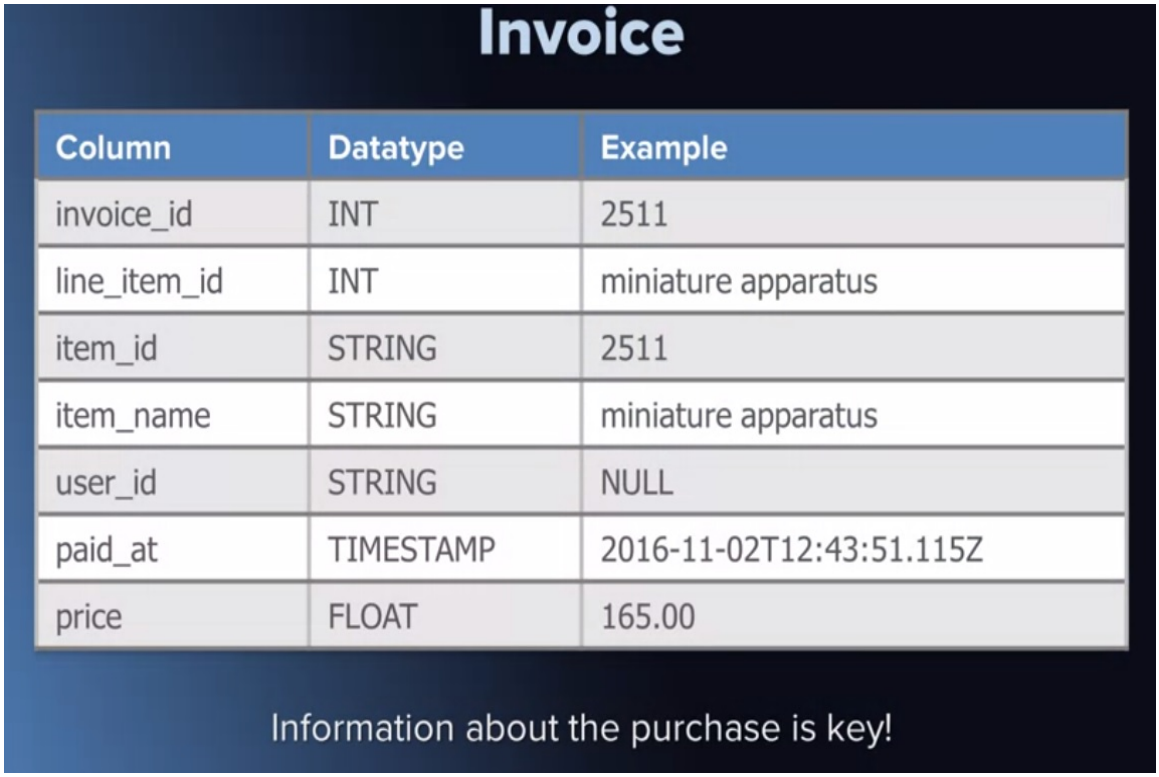

Invoice
Column
Datatype
Example
invoice_id
INT
2511
line_item_id
INT
miniature apparatus
item_id
STRING
2511
item_name
STRING
miniature apparatus
user_id
STRING
NULL
paid_at
TIMESTAMP
2016-11-02T12:43:51.115Z
price
FLOAT
165.00
Information about the purchase is key!

Figure 2.3

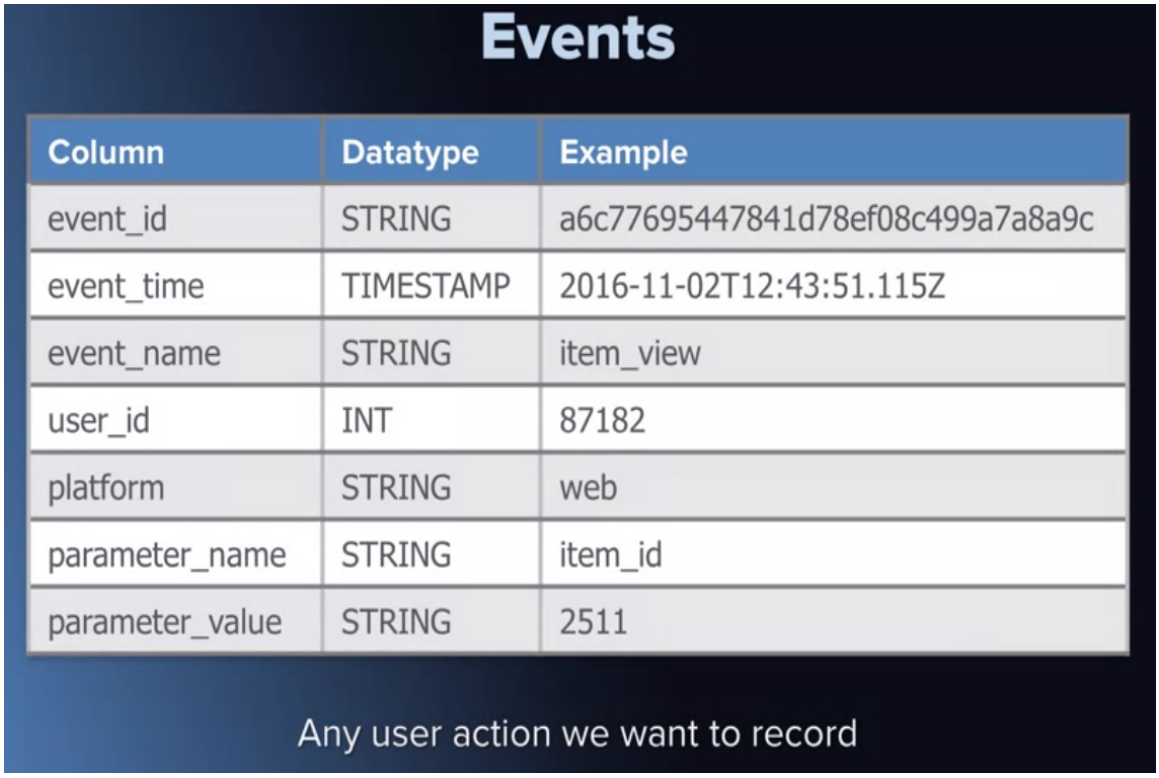


Figure 2.4: The event_id is in the form of a universally unique identifier. There are some advantages to this compared to using just using an integer. Initially the event_id looks looks like a primary id. It isn't however because there could be multiple rows with the same event_id.

2.1.2 Data Wrangling & Flexible Data Formats

The tables above represent the types of tables we will be working with in this course. Probably the most difficult table to work with is the events table. Here we will discuss what an even table is trying to accomplish. Along with the concept of pivoting data.

Events tables

An event table can be thought of as a receipt of real things that happened. Generally these tables are not edited or updated once they are created. They are different compared to something like a Users table because they are quite large and they don't need to be stored in a specific format. If you want to learn more about how different types of data are stored, you can look into the concept data warehousing.

An example of an event table can be found in Figure 2.4. Initially the event_id looks looks like a primary id. It isn't however because there could be multiple rows with the same event_id. Let's consider a specific event in the event table i.e. "event_name=view_user_profile".

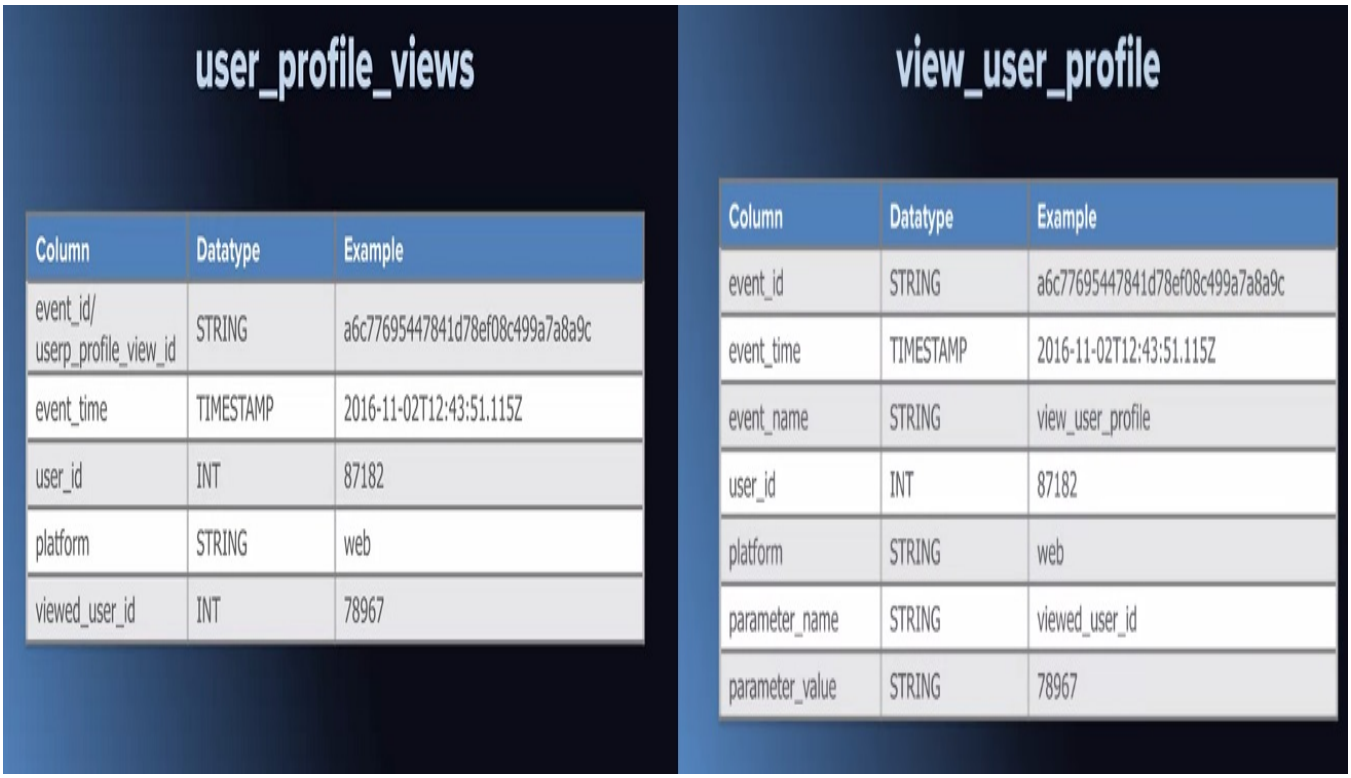
For analysis we may want to know separate things about the actions a user can take. Some of these actions are characterized in Figure 2.5. The caption describes the process of transferring this feature into separate categories and then writing it in a flexible format. Ideally, if we only get one possible format, it needs to be flexible enough for us and others to use.

As an example, we will show how to do this in SQL for the "view_user_profile" event in the code below.

Listing 2.1: Writing the view_item event in a flexible format.

```
SELECT event_id,
    event_time,
    --event_name,
    user_id,
    platform,
    parameter_value AS viewed_user_id
FROM
    dsv1069.events --data file for this table
WHERE
    event_name = 'view_user_profile'

--What could possibly go wrong with this?
--What could make the events table an incorrect source?
--Is some of the data missing values?
```



user_profile_views

Column	Datatype	Example
event_id/ userp_profile_view_id	STRING	a6c77695447841d78ef08c499a7a8a9c
event_time	TIMESTAMP	2016-11-02T12:43:51.115Z
user_id	INT	87182
platform	STRING	web
viewed_user_id	INT	78967

view_user_profile

Column	Datatype	Example
event_id	STRING	a6c77695447841d78ef08c499a7a8a9c
event_time	TIMESTAMP	2016-11-02T12:43:51.115Z
event_name	STRING	view_user_profile
user_id	INT	87182
platform	STRING	web
parameter_name	STRING	viewed_user_id
parameter_value	STRING	78967

Figure 2.5: Further analysis may require us to subdivide categories for certain features. The table labeled ”user_profile_views” shows us how the event_name=view_user_profile can be given features relevant to that topic. Having something like a ”viewed_user_id” is rather redundant though and unintuitive to someone not as familiar with SQL. If we only have one format, it needs to be flexible enough for people to work with. This is why its better to create a table that is reminiscent of the events table given in Figure 2.4. This is illustrated in the table ”view_user_profile”.

```
--Check that all the days have data.
--Check when we started recording the event.

SELECT event_id,
    event_time,
    user_id,
    platform,
    CAST(parameter_value AS INT) AS viewed_user_id
-- The CAST function converts data from one type to another.
--It takes the form of CAST(expression AS [dataform])
FROM
    dsv1069.events --data file for this table
WHERE
    event_name = 'view_user_profile'

--The reason we use the CAST function is so that we can
--do math on a string based value.
--How do we extend this if we want to have more parameter names
--as columns in a well organized manner.
--Here is one way to do that.

SELECT event_id,
    event_time,
    user_id,
    platform,
    --CAST(parameter_value AS INT) AS viewed_user_id,
    (CASE WHEN paramater_name == 'viewed_user_id'
        THEN CAST(parameter_value AS INT)
        ELSE NULL
            END) AS viewed_user_id
FROM
    dsv1069.events --data file for this table
WHERE
    event_name = 'view_user_profile'

--Something that could go wrong here is that a parameter might
--not be listed for all the events.
```

```
--Solution to the 'view_item' name
SELECT event_id,
  event_time,
  user_id,
  platform,
  MAX(CASE WHEN parameter_name = 'item_id'
    THEN CAST(parameter_value AS INT)
    ELSE NULL
      END) AS item_id,
  MAX(CASE WHEN parameter_name = 'referrer'
    THEN parameter_value
    ELSE NULL
      END) AS referrer
FROM dsv1069.events
WHERE event_name = 'view_item'
GROUP BY event_id,
  event_time,
  user_id,
  platform

--Without the aggregate function the table generates too many rows.
--The CASE statement is creating two rows for each event_id.
--The MAX function fixes this by picking the MAX value
--between a NULL and an integer.
--When MAX is applied to an string its only meaningful when
--one of the elements is NULL.
--In that case it will reject the NULL value.
```

2.1.3 Identifying Unreliable Data & Nulls

Imagine you are new to a database and you are asked a question with interlinking tables in the schema. How do you find the right data for the problem along with trusting its reliability? After this section, we will be able to segment and visualize data to check for completeness along with identify any data that might be missing. Here are some simple checks when analyzing an unknown database.

- Consider the name of the table:
  - If it has a date then it might not be relevant to the current time.

  - Look at the schema of the data i.e. name before the dot or function is applied. It might indicate that it was made for a specific purpose.

- Use existing information:
  - Information may pertain to a specific event and it can come from SQL.
  - If the data already exists as a table, it might be more efficient to use it in its original form.

- Events are built, not born:
  - Each event is implemented by hand, and its type is designed personally by a developer.
  - Just because you don't see an event occurring, it doesn't mean that it didn't happen, rather it wasn't recorded.

- A good sanity to check is to see if the table is empty or not with a simple GROUP BY statement to check COUNT(*). Check to see if the number of rows is what you would expect.

NULL values

NULL values can be quite sneaky when disrupting the results from a data set.

Listing 2.2: Dealing with NULL values.

```
--CODE 1--
SELECT name
FROM example_table et
JOIN other_example_table oet
ON oet.user_id = et.user_id
AND et.user_id IS NOT NULL
--END CODE 1--

--One possible way to deal with this may be to exclude the NULLS.
--An example of this is given as

SELECT *
FROM example_table et
JOIN other_example_table oet
ON oet.user_id = et.user_id
```

```
--Other times you may need a more delicate touch such as

SELECT COALESCE(parent_user_id, user_id) AS original_user_id,
--Syntax: COALESCE(val1, val2, ...., val_n)
--The COALESCE() function returns the first non-NULL value in a list.
parent_user_id,
user_id
FROM dsv.users

--Here we are replacing a value when it is NULL.
--Always be skeptical when moving data from one form to another.
```

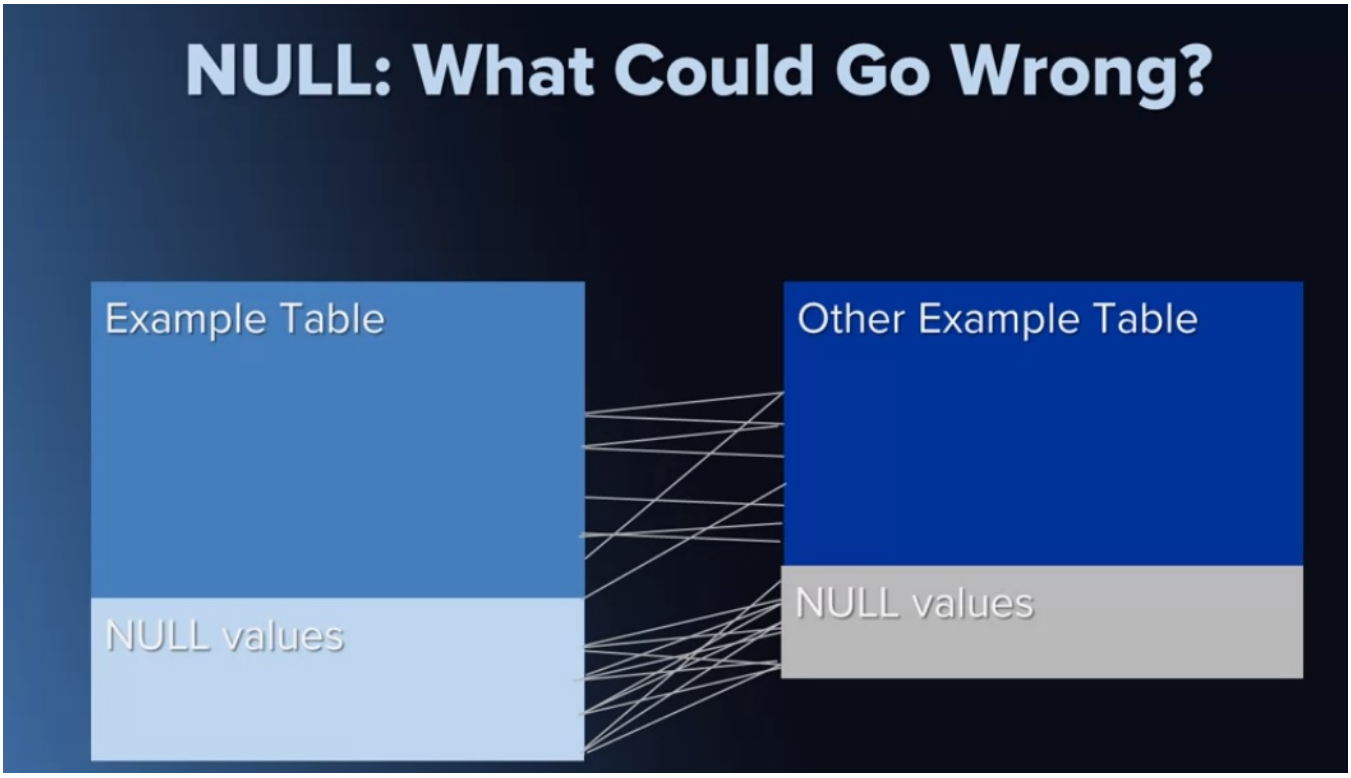


Figure 2.6: Each of these white lines represent how the join function maps the keys from one table to the next. Many of the keys will be fine but others will misrepresent the data you want to work with. Notice how the NULL values will match each other from both tables. This idea can really get out of hand. This diagram represents code that is written in Listing 2.2 under CODE 1

Listing 2.3: Working through exercises to show a table which tells us how many user were created, how many were deleted, how many were merged and the net number of created users.

```
SELECT
  new.day,
  new.new_users_created,
  COALESCE(deleted.deleted_users,0) AS deleted_users,
  COALESCE(merged.merged_users,0)   AS merged_users,
  (new.new_users_created-COALESCE(deleted.deleted_users,0)-COALESCE(merged.merge
FROM
  (SELECT date(created_at) as day,
  COUNT(*) AS new_users_created
  FROM dsv1069.users
  GROUP BY date(created_at)) new
LEFT JOIN
  (SELECT
  date(created_at) AS day,
  COUNT(*) AS deleted_users
  FROM
  dsv1069.users
  WHERE
  deleted_at IS NOT NULL
  GROUP BY date(created_at) ) deleted
ON deleted.day = new.day
LEFT JOIN
  (SELECT
  date(created_at) AS day,
  COUNT(*) AS merged_users
  FROM
  dsv1069.users
  WHERE
  (id <> parent_user_id OR parent_user_id IS NOT NULL)
  GROUP BY date(created_at) ) merged
ON merged.day = new.day
ORDER BY day
```

2.2 Week 2

2.2.1 Data Types Supported by SQL

The different data types that are supported by SQL are given in figures 2.7, 2.8, 2.9.

Numeric Data Types

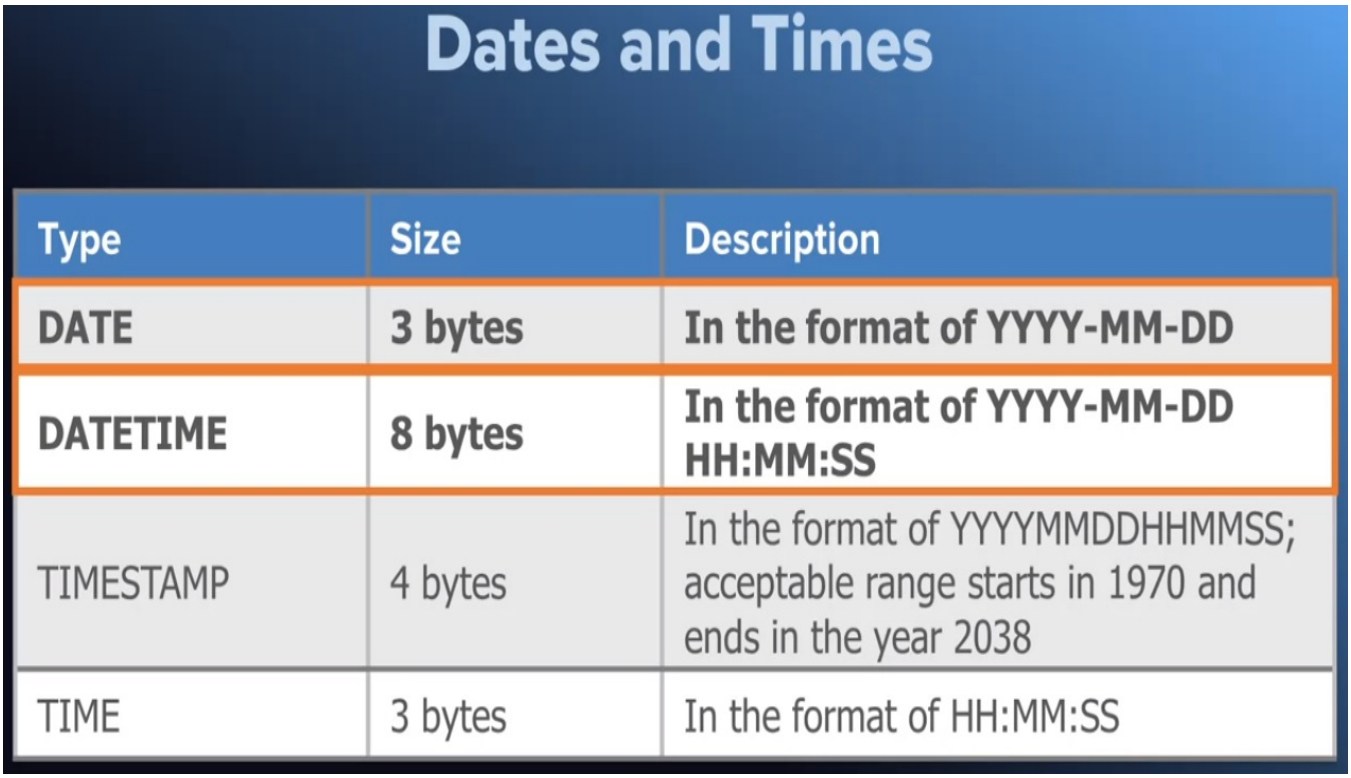
Type	Size	Description
TINYINT[Length]	1 byte	Range of –128 to 127 or 0 to 255 unsigned
SMALLINT[Length]	2 bytes	Range of –32,768 to 32,767 or 0 to 65,535 unsigned
MEDIUMINT[Length]	3 bytes	Range of –8,388,608 to 8,388,607 or 0 to 16,777,215 unsigned
INT[Length]	**4 bytes**	**Range of –2,147,483,648 to 2,147,483,647 or 0 to 4,294,967,295**
BIGINT[Length]	**8 bytes**	**Range of –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 or 0 to 18,446,744,073,709,551,615 unsigned**
FLOAT[Length, Decimals]	4 bytes	A small number with a floating decimal point
DOUBLE[Length, Decimals]	8 bytes	A large number with a floating decimal point
DECIMAL[Length, Decimals]	Length + 1 or 2 bytes	A DOUBLE stored as a string, allowing for a fixed decimal point

Figure 2.7: Numeric data structures supported by SQL. The most commonly used ones will be INT, BIGINT, and DOUBLE.

Text Data Types

Type	Size	Description
CHAR[Length]	**Length bytes**	**A fixed-length field from 0 to 255 characters long**
VARCHAR[Length]	String length + 1 or 2 bytes	A variable-length field from 0 to 65,535 characters long
TINYTEXT	String length + 1 bytes	A string with a maximum length of 255 characters
TEXT	String length + 2 bytes	A string with a maximum length of 65,535 characters
MEDIUMTEXT	String length + 3 bytes	A string with a maximum length of 16,777,215 characters
LONGTEXT	String length + 4 bytes	A string with a maximum length of 4,294,967,295 characters

Figure 2.8: String data structures supported by SQL. The most commonly used one is CHAR but it might not support enough characters.



Dates and Times

Type	Size	Description
DATE	**3 bytes**	**In the format of YYYY-MM-DD**
DATETIME	**8 bytes**	**In the format of YYYY-MM-DD HH:MM:SS**
TIMESTAMP	4 bytes	In the format of YYYYMMDDHHMMSS; acceptable range starts in 1970 and ends in the year 2038
TIME	3 bytes	In the format of HH:MM:SS

Figure 2.9: Date and time data structures supported by SQL. The DATE or DATETIME format are the most common forms found.

2.2.2 What is a Dependency?

Real data sets used in industry are often not accurately represented in course content. A big example of this is how data sets are changing all the time in the real world. The data sets used in this course are not that large and many times, we need to conserve computational time when analyzing data.

A way to deal with computational time is to alter the content of a table rather than recreating it. From there, we can calculate new result and modify it based on specific dates. We can do this in whats called an Automated Query in that table construction is automated based on desired outcome. Granted you can't iterate in SQL, you can still write a program which generates the results your looking for and allocate a date to it.

As an example, in the previous module, we created a view-item table which listed new data values that we desired. With that program, we can automate it so that a new view-item table will be constructed after new data comes in (thus simulating real world data). There is more to it than just slapping on dates with the results. We need to make sure the other tables that are used are properly updated as well. This process is what we call a pipeline. The pipeline from the previous module would be

Pipeline: Events $\rightarrow$ view_item events table $\rightarrow$ Most Recently Viewed Item

If we correctly construct all the tables to automate in this manner, then we have constructed a data pipeline with multiple dependencies. A commonly used term to describe the steps happening in this table creation is Extract Transform Load (ETL).

One thing to consider when constructing a pipeline is to think about previous data that are not properly dated. This process of filling up past tables with dates is called Backfilling and it is a very large field in data engineering.
Definition: Dependency - When the data in query refers to data in a preceding table.
Definition: Stale - When the data in a table doesn't reflect the most up-to-date information.
Definition: ETL - Extract Transform Load.
Definition: Job - The task given to a database to perform ETL.
Definition: Pipeline - Several table creation/update tasks in a specific order. Definition: Backfill - To run a table creation/update task on a rage of dates in the past.

2.2.3 Create a View-Item Table

Here we will create a table based on the events stream which in itself will not be data analysis, rather it will be an analytical tool. The raw events stream table is represented in Figure 2.10. When starting with a source, this would be the time to filter and clean and data that is going to be used in the pipeline.

Raw Events Stream

	event_id	event_time	user_id	event_name	platform	parameter_name	parameter_value
1	b9de71c5c3cc4cd7a97e50b832106e5a	2017-06-26T11:23:39.244242	178481	view_item	android	item_id	3526.0
2	23267713c9ea44419331731f50b6a8db	2017-06-27T10:46:39.244242	178481	view_item	android	item_id	1514.0
3	1b7822fa7b854e01970218ae8f721fe0	2017-06-27T11:15:39.244242	178481	view_item	android	item_id	3712.0
4	2a7a188a626841ac94befcc419f06af4	2016-10-05T20:43:10.244242	154133	view_item	android	item_id	3586.0
5	631d657264cc4616a4528f759509b25d	2016-10-04T03:29:10.244242	154133	view_item	android	item_id	1061.0

Figure 2.10: Table illustrating the content in the Raw_events stream.

Some forms of filtering and cleaning can include removing events that are generated while testing internally, trimming long-tail categorical data, and replacing NULL values with appropriate values. Cleaning the data is quite important as it will make it easier to read future queries and in collaborating with others. It also improves the efficiency since you will only need to compute it once. Lastly, it provides standardization so that anyone who might be working with the data can easily understand what has been done there. Some things to consider when cleaning the data are the Upstream dependencies (starting data set) and the Downstream dependencies (tables that haven't been built yet).

Definition: Upstream Dependency - A task or table that happens before the final outcome of the pipeline.
Definition: Downstream Dependency - A task or table that that must be delivered before something else can start from the pipeline.

2.2.4 Heirarchy of Data

There are many moving pieces that need to be working for good data science to happen. Many assumptions aren't stated but they need to be recognized. For data science a hierarchy of concepts is presented in figure 2.11. In this course, we followed this hierarchy as

1. Collect - Generated the data with python scripts and random number generator.
    - In reality, data would be coming from real live users.
2. Move/Store - This data was uploaded and stored in Mode which is a nontrivial amount of work.
    - In reality, data would be processed into an analytics database.
3. Aggregate and Label data - We practiced this when we counted the users added each day.
4. Data Insight - We did this when we counted the users each day as we could create some insight.
5. Explore/Transform - In the last two module, we will work on concepts in this area.

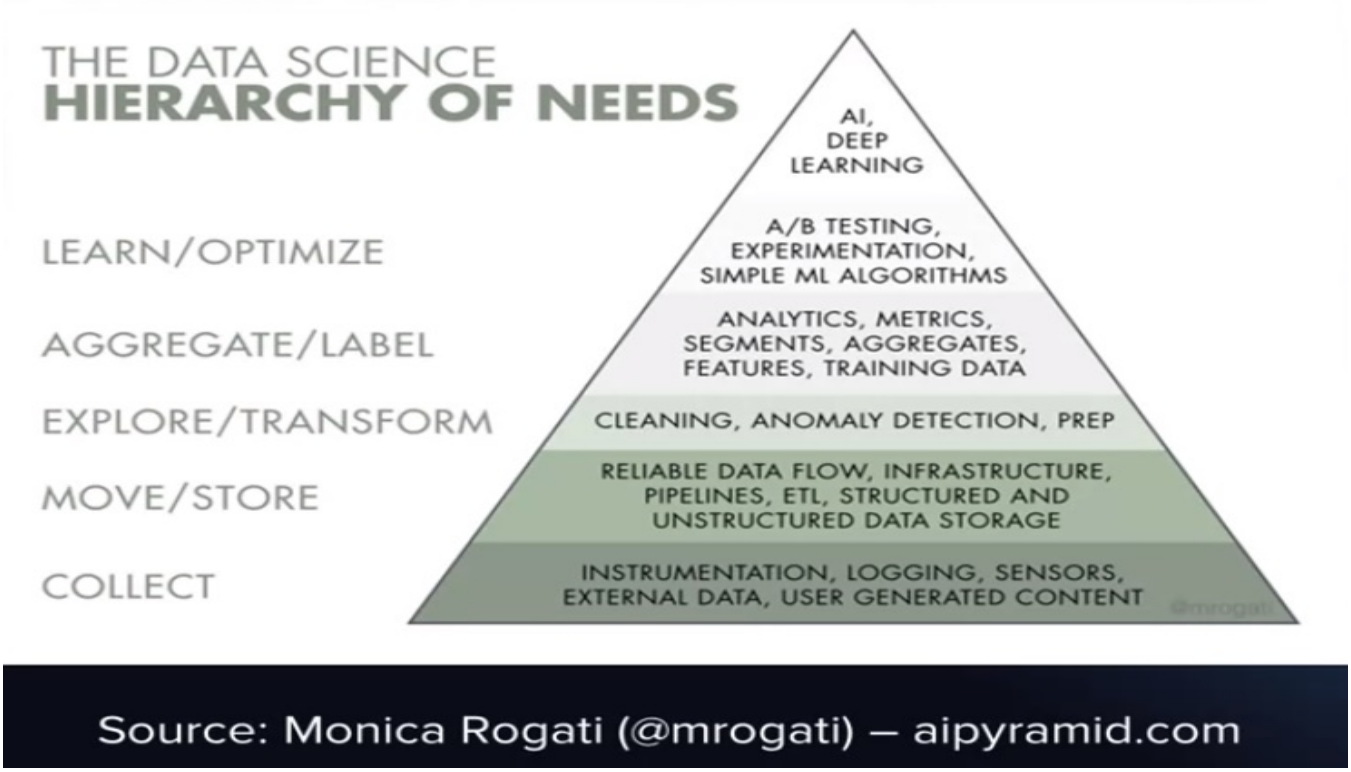


Figure 2.11: Illustration showing how concepts in data science depend on one another.

2.2.5 Create User Info Table

When we were counting the number of users we had, it was quite tedious doing this. Let's try and automate some of this work through a pipeline. The columns we want to add from our previous table are: created_today, Is_deleted, deleted_today, has_ever_ordered, ordered_today, and date.

From here, lets determine whether our dependencies are going to be upstream or downstream as
Upstream: Users table, Orders table.
Downstream: Dashboard tables, Prediction scores computed using these features. We need to ask certain questions like how will we build the table if its going to be updated daily? This means we will want a query which can add to the table with the latest updates. When using this table, it will need to work for dates in the past i.e. with Backfilling.

Listing 2.4: Creating a User Snapshot table .

```
{% assign ds = '2018-01-01' %}--This is essentially creating a constant date as a
SELECT id, '{{ds}}' AS variable_column
FROM users
WHERE created_at <= '{{ds}}'

--This could be used to declare a parameter in the query

{% assign ds = '2018-01-01' %}
INSERT INTO user_info
--This line links the data generated from this query
--into the table constructed in the query below.
SELECT
id                                                          AS user_id,
IF(users.created_at = {{ds}}, 1, 0)                         AS created_today,
IF(users.deleted_at <= {{ds}}, 1, 0)                        AS is_deleted,
IF(users.deleted_at = {{ds}}, 1, 0)                         AS is_deleted_today,
IF(users_with_orders.user_id IS NOT NULL, 1, 0)             AS has_ever_ordered,
IF(users_with_orders_today.user_id IS NOT NULL, 1, 0)       AS ordered_today,
'{{ds}}'                                                    AS ds
FROM users
LEFT JOIN
(SELECT DISTINCT user_id
    FROM orders
```

```
    WHERE created_at <= '{{ds}}'
) users_with_orders
ON users_with_orders.user_id = users.id
LEFT JOIN
(SELECT DISTINCT user_id
FROM orders
WHERE created_at = '{{ds}}'
) users_with_orders_today
ON users_with_orders_today.user_id = users.id
WHERE users.created_at <= '{{ds}}'

--You can dissect this query a bit more but it is
--essentially creating data from the users and orders table,
--with specific time stamps relative to '{{ds}}'.

--Below we are creating a new table from the data
--we wrangled above.

CREATE TABLE IF NOT EXISTS user_info
--This statements creates the table if non exist already
--and calls it user_info
(
    user_id             INT(10) NOT NULL,
    created_today       INT(1)  NOT NULL,
    is_deleted          INT(1)  NOT NULL,
    is_deleted_today    INT(1)  NOT NULL,
    has_ever_ordered    INT(1)  NOT NULL,
    ordered_today       INT(1)  NOT NULL,
    ds                  DATE    NOT NULL
);
DESCRIBE user_info
```

2.3 Week 3

2.3.1 Introduction to SQL Problem Solving

Data science has many tools in their arsenal. SQL is a very fundamental tool that works very well at exploring/moving data around. SQL joins are only a small part of SQL. Their proper use requires planning and design skills along with a strong familiarity with the data set. This section will focus on practicing our SQL skills on data related problems. The goal of this section is to map out joins, be able to highlight the level of detail needed for different kinds of questions, practice answering data related questions, develop strategies for answering questions.

2.3.2 Mapping out Join Statements

The first thing we need to do is to identify the question we are trying to answer. Lets consider the following question, the columns, and the tables we will need to answer it.

Question: What percent of users have viewed items from each product category?

Columns: Category, Count of users, Need a total user number.

Tables: Viewed Events (to get user ids and item_ids), Items (to get category for each item_id), and Users Table (to get a total user count).

We built the appropriate subqueries for the Viewed-Item events in Module 1 through Listing 2.3 and we created the table in Listing 2.4. The total users might be a subquery. The methodology presented above is shown in the Listing 2.5.

Listing 2.5: Test Joins .

```
--First figure out the right order for the Join statement

SELECT *
FROM viewed_items
JOIN items
ON items.id = viewed_items.items_id

--Next we should add in the columns

Select
```

```
    Items.category,
    COUNT(DISTINCT user_id) AS users_viewed_count
FROM viewed_items
JOIN items
ON items.id = viewed_items.item_is
GROUP BY Items.category

--Clean up the formatting if need be.
--Make sure aggregate functions are labeled names so that
--someone reading it can understand what it is for.
```

2.3.3 Test Queries vs. Final Queries

We will consider the following problem when working through how to use test queries.

Question: Which users have ever ordered a widget?

Columns: Count of Users.

Tables: Orders.

The work for this is done in Listing 2.6.

Listing 2.6: Test Joins .

```
--Quick Queries

SELECT COUNT(DISTINCT user_id)
FROM ORDERS
WHERE item_category = 'widget'

--This is used to see something quickly, not as a final outcome for the query.

SELECT
    orders.item_category,
    COUNT(DISTINCT COALESCE(parent_user_id,user_id) AS users_with_orders
FROM
    ORDERS
JOIN
```

```
    Users
ON
    Users.id = orders.user_id
GROUP BY
    orders.item_category

--This is what a final query might look like as it is easy to read
--and is shareable to others.
```

2.3.4 Example: Rolling Table

Listing 2.7: Creating a rolling table to answer the question: How many orders are there per day with a rolling order?.

```
--Exercise 1:Create a subtable of orders per day.
--Make sure you decide whether you are counting invoices or line items
--Definition: a rolling order is the sum/average seven days behind the purchase d

SELECT
    COUNT(DISTINCT invoice_id) AS  num_invoice,
    COUNT(DISTINCT line_item_id) AS num_line_item,
    date(paid_at) AS day
FROM
    dsv1069.orders
GROUP BY
    day

--Since there might be some dates missing, --we need to join this into a date rol

--Exercise 2: Check your joins. We are still trying to
--count orders per day.
--In this step join the sub table from the previous exercise to
--the dates rollup table so we can get a row for every date.
--Check that the join works by just running a SELECT * query.

SELECT *
FROM
  dsv1069.dates_rollup
```

```
LEFT JOIN
(SELECT
  COUNT(DISTINCT invoice_id) AS num_invoice,
  COUNT(DISTINCT line_item_id) AS num_line_item,
  date(paid_at) AS day
FROM
  dsv1069.orders
GROUP BY
  day
) items_per_day
ON items_per_day.day=dates_rollup.date

--dates_rollup table is a new table which records dates that are 7 days
--before date and 28 days before date.
--We can make a key for join from the date() function.

--Exercise 3: Clean up your Columns. In this step be sure to specify
--the columns you actually want to return, and if necessary do any
--aggregation needed to get a count of the orders made per day

SELECT
  dates_rollup.date,
  COALESCE(SUM(num_invoice),0) AS orders,
  COALESCE(SUM(num_line_item),0) AS items_ordered
FROM
  dsv1069.dates_rollup
LEFT JOIN
(SELECT
  COUNT(DISTINCT invoice_id) AS num_invoice,
  COUNT(DISTINCT line_item_id) AS num_line_item,
  date(paid_at) AS day
FROM
  dsv1069.orders
GROUP BY
  day
) items_per_day
ON
    items_per_day.day=dates_rollup.date
```

```
GROUP BY
  dates_rollup.date

--Here we are counting the sum of the invoices as the number of orders.
--Likewise, we are counting the sum of the items ordered as the number of
--items ordered.
--We need the coalesce function to fill in zeroes for NULL values.

--Exercise 4: Weekly Rollup. Figure out which parts of the JOIN condition
--need to be edited to create a 7 day rolling orders table.

SELECT
  dates_rollup.date,
  COALESCE(SUM(num_invoice),0) AS orders,
  COALESCE(SUM(num_line_item),0) AS items_ordered,
  COUNT(*) AS rows
FROM
  dsv1069.dates_rollup
LEFT JOIN
(SELECT
  COUNT(DISTINCT invoice_id) AS num_invoice,
  COUNT(DISTINCT line_item_id) AS num_line_item,
  date(paid_at) AS day
FROM
  dsv1069.orders
GROUP BY
  day
) items_per_day
ON
  dates_rollup.date >= items_per_day.day
AND
  dates_rollup.d7_ago < items_per_day.day
GROUP BY dates_rollup.date

--Here we are making it so that it sums all the orders and
--all the items ordered over that week.
```

2.3.5 Realistic Case study

Suppose the marketing team has this idea. We should email people a picture of what they looked at most recently. They might have an idea of what they want the website to look like (called a wireframe). For each of these, we will need tables constructed as in Table 2.1.

Listing 2.8: SQL code to answer the question, we should email people a picture of what they looked at most recently .

```
--First step, create the right subtable for recently viewed events
--using the view_item_events table.

SELECT
  user_id,
  item_id,
  ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY event_time ASC)
  AS view_num
  --The ROW_NUMBER, RANK, and DENSE_RANK functions all do the same thing here.
FROM
  dsv1069.view_item_events

--Second step, check your JOINS.

SELECT *
FROM
(
```

Columns	Tables	Restrictions
user_id	user/view_items	Only users who viewed items
email_address	users	
item_id	view_items/items	Only items MOST RECENTLY viewed
item_name	items	
item_category	items	
Subtable		
user_id	users\view_items	Only users who viewed items
item_id	view_items/items	
view_rank	computed	Rank order of which was viewed most recently

Table 2.1: Building a wireframe to answer the question: We should email people a picture of what they looked at most recently.

```
SELECT
  user_id,
  item_id,
  ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY event_time ASC)
  AS view_num
FROM
  dsv1069.view_item_events
) recent_views
JOIN
  dsv1069.users
ON
  users.id=recent_views.user_id
JOIN
  dsv1069.items
ON
  items.id=recent_views.user_id

--Third step, clean up the columns you want to use

SELECT
  users.first_name                                    AS first_name,
  users.last_name                                     AS last_name,
  recent_views.user_id                                 AS user_id,
  recent_views.view_num                               AS view_num,
  recent_views.item_id                                AS item_id,
  users.email_address,
  items.name                                          AS item_name,
  items.category                                      AS item_category
FROM
(
SELECT
  user_id,
  item_id,
  ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY event_time ASC)
  AS view_num
FROM
  dsv1069.view_item_events
) recent_views
```

```
JOIN
  dsv1069.users
ON
  users.id=recent_views.user_id
JOIN
  dsv1069.items
ON
  items.id=recent_views.user_id

--Is there any extra filtering that we want?

--What is a user has been deleted or merged?
--It would be weird to send them an email when they might have already seen it.

SELECT
  users.first_name                                          AS first_name,
  users.last_name                                           AS last_name,
  COALESCE(users.parent_user_id,recent_views.user_id)
AS user_id,
  recent_views.view_num                                     AS view_num,
  recent_views.item_id                                      AS item_id,
  users.email_address,
  items.name                                                AS item_name,
  items.category                                            AS item_category
FROM
(
SELECT
  user_id,
  item_id,
  ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY event_time ASC)
  AS view_num
FROM
  dsv1069.view_item_events
) recent_views
JOIN
  dsv1069.users
ON
  users.id=recent_views.user_id
```

```
JOIN
  dsv1069.items
ON
  items.id=recent_views.user_id

--will answer this problem.

--We could only look at data after a certain point by putting in
--WHERE view_item_events.event_time >='2017-01-01'

--We could filter out any deleted accounts by adding
--WHERE users.deleted_at IS NULL
--for the entire query.

--Maybe we don't want to send people a notification
--if they already bought that thing.

SELECT
  users.first_name                                    AS first_name,
  users.last_name                                     AS last_name,
  COALESCE(users.parent_user_id,recent_views.user_id)
AS user_id,
  recent_views.view_num                               AS view_num,
  recent_views.item_id                                AS item_id,
  users.email_address,
  items.name                                          AS item_name,
  items.category                                      AS item_category
FROM
(
SELECT
  user_id,
  item_id,
  ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY event_time ASC)
  AS view_num
FROM
  dsv1069.view_item_events
) recent_views
JOIN
```

```
    dsv1069.users
ON
    users.id=recent_views.user_id
JOIN
    dsv1069.items
ON
    items.id=recent_views.user_id
--NEW STUFF--
LEFT JOIN
      dsv1069.orders
ON
      orders.item_id = recent_views.item_id
AND
      orders.user_id = recent_views.user_id
WHERE
    view_num = 1
AND
      users.deleted_at IS NOT NULL
AND
      orders.item_id IS NULL
```

Windowing functions

Listing 2.9: Showing how windowing functions are made, the syntax for some of them, and what their function is .

```
SELECT
      user_id,
      invoice_id,
      paid_at
FROM
      dsv1069.orders
GROUP BY user_id
--This GROUP BY statement won't work because we aren't
--using any aggregate functions.
ORDER BY
paid_at
```

```
--SQL Server Window Functions calculate an aggregate value
--based on a group of rows and return multiple rows for each group.

SELECT
    user_id,
    invoice_id,
    paid_at,
    RANK(  ) OVER (PARTION BY user_id ORDER BY paid_at ASC) AS order_num,
    --RANK ( ) OVER ( [ partition_by_clause ] order_by_clause )
    --RANK() Assigns a rank value to each row within a
    --partition of a result set
    --The rank of a row is one plus the number of ranks that
    --come before the row in question.

    --partition_by_clause divides the result set produced by the
    --FROM clause into partitions to which the function is applied.

    --order_by_clause determines the order of the data
    --before the function is applied.
    DENSE_RANK( ) OVER (PARTIION BY user_id ORDER BY paid_at ASC) AS
    dense_order_num,
    --DENSE_RANK() Assign a rank value to each row within a partition
    --of a result, with no gaps in rank values.
    ROW_NUMBER( ) OVER (PARTITON BY user_id ORDER BY paid_at ASC) AS row_num
    --ROW_NUMBER() assigns a unique sequential integer to rows within a
    --partition of a result set, the first row starts from 1.
FROM
    dsv1069.orders

--The way to answer this problem is to use a DENSE_RANK() call
--since we want to window each result without gaps.

SELECT *
FROM
(
    SELECT
        user_id,
        invoice_id,
```

```
        paid_at,
        RANK(   ) OVER (PARTION BY user_id ORDER BY paid_at ASC)
        AS order_num,
        DENSE_RANK( ) OVER (PARTIION BY user_id ORDER BY paid_at ASC)
        AS dense_order_num,
        ROW_NUMBER( ) OVER (PARTITON BY user_id ORDER BY paid_at ASC)
        AS row_num
    FROM
        dsv1069.orders
) sub
WHERE order_num=1
--We need to do this outside the query since the column
--we want doesn't exist outside of the query.
```

2.3.6 Product Analysis

Some basic questions we can answer through SQL are:

How many times do users re-order a specific item?

How many time do users re-order an item from a specific category?

How long between orders/re-orders?

Answering these types of questions are important for companies and here are some reason as to why.

1. Maybe we want to improve recommendations for each user.
    - These might be products that a user orders all the time or sound appealing to someone who hasn't ordered from here in a while.
2. We want to understand if a user's purchase history can help us make better recommendations.
    - We want to understand our customers behaviour towards these products.
3. We want to pick a single strategy to build and test.

These kinds of ideas can lead to better questions to ask. An idea about the thought process occurring when using SQL can be found in 2.10

Listing 2.10: Order the logic towards problem solving in SQL .

```
--How many users do we have?
SELECT COUNT(DISTINCT id)
FROM dsv1069.users

--How many users have ever ordered?

SELECT COUNT(DISTINCT user_id)
FROM dsv1069.orders

--Goal find how many users have reordered the same item
SELECT
  COUNT(DISTINCT user_id) AS users_who_ordered
FROM
(
SELECT
  user_id,
  item_id,
COUNT(DISTINCT line_item_id) AS times_user_ordered
FROM
  dsv1069.orders
GROUP BY
  user_id,
  item_id
) user_level_orders
WHERE
  times_user_ordered >1

--Do users even order more than once?
SELECT COUNT(DISTINCT user_id)
FROM
(
  SELECT
    user_id,
    COUNT(DISTINCT invoice_id) AS order_count
  FROM dsv1069.orders
  GROUP BY
```

```
    user_id
) user_level
WHERE order_count>1

--Orders per item

SELECT
  item_id,
  COUNT(line_item_id) AS times_ordered
FROM dsv1069.orders
GROUP BY
  item_id

--Items per category
SELECT
  item_category,
  COUNT(line_item_id) AS times_ordered
FROM dsv1069.orders
GROUP BY
  item_category

--Do users order multiple items from the same category

SELECT
  item_category,
  AVG(times_category_ordered) AS avg_times_category_ordered
FROM
(
  SELECT
    user_id,
    item_category,
    COUNT(DISTINCT line_item_id) AS times_category_ordered
  FROM
    dsv1069.orders
  GROUP BY
    user_id,
    item_category
) user_level
```

```
GROUP BY item_category
ORDER BY avg_times_category_ordered DESC
```

2.4 Week 4: A/B Hypothesis Testing

Within this course, we worked on understanding dirty data sets (week 1), how to create and maintain tables (week 2), and how SQL can be used as a problem solving tool (week 3). Now we are going to apply SQL in a basic A/B hypothesis testing scenario. We will test it using causal inference.

2.4.1 Statistics refresher

Having a strong understanding of statistics is important in data science as it is very fundamental in creating realistic predictions from data. In order to refresh ourselves on some of the relevant topics, lets work on a specific problem. Consider the question

Question: How many seats are there on the train?

Better Question: What is the average number of seats available on the train?

The best way to answer this problem is by having many observations of the event. In doing so, we could approximate the distribution as a normal distribution through the central limit theorem and then create a confidence interval based on the sample set.

A/B testing

Consider the case where two people are analyzing the number of people on the train. If both of those people have a different number of observations, is this difference in the number of observations a problem? Who's estimate is more accurate? In general, the more observations one makes, the better the estimate will be and thus the stronger statement we can make. Many times these questions are quantified through a confidence interval.

One of the key steps in calculating and A/B hypothesis test is through the standard error, i.e.

$$\text{standard error} = \frac{\sigma}{\sqrt{n}}.$$

Assuming a normal distribution, we can use the Z-score to determine a confidence interval.

The key idea with A/B testing is to compare two hypothesis tests with one another and find if there is a statistically significant difference between options A and B. As an example, maybe the question we asked above isn't the best question to be asking. Consider problems A and B:

Problem A: How many seats are there on the train?

Problem B: How likely is it that I will have a seat on the train?

In reality, I only care about answering problem B since its the only thing that matters to me on my commute. But the interesting this to answer is, how much better is problem B compared to problem A, and that is what we will answer using SQL.

Definition: A/B Hypothesis Testing A method of comparing two statistical tests to one another to determine which one performs better.

2.4.2 Test Assignments

Let's try to understand what happens when a user is placed in an experiment. Maybe we want to develop some new software features and we would like to know what eligibility criteria need to be met for this software feature.

Question: What happens when a user is put into an experiment?

Some software features are aimed at a particular situation which means some eligibility criteria must be met. In addition to this eligibility criteria, the user must perform an action such as: a user visiting a website or that user making multiple purchases. When an eligible user triggers this new feature, we call this a diversion point and at that point we will include them in the experiment. When the user triggers a diversion point, we call this a treatment.

What information do we need to record for this event? The treatment event should have the columns below

Columns: Event_id, Event_time, Event_name, User_id, Parameter_name, Parameter_value.

While we are developing this treatment, we can create a control group to compare the results of implementing this new feature to the prexisting feature. This will in essence create an A/B test where we can compare how well our new treatment is to the previous system.

Case study 1: User Welcome Email

Imagine we already send a welcome email when a user create an account. We would like to update this feature to show images of top selling items.

Eligibility: Has the user created an account with a valid email address?

Diversion Point: When is the automated email scheduled to be sent?

Control: Gets existing email.

Treatment: Gets new email with top selling items.

Engagement Metrics: Email opens, email clicks, orders placed, total number of orders placed, and the revenue.

Case Study 2: Push Notifications to Mobile Users.

We would like to send a push notification to users, if they have items in their cart, and they have not completed an order after 24 hour. They should only receive this push notification if they haven't received a push notification in the last two weeks.

Eligibility: Created an account with a valid email address, users with a view item event on a mobile device who have added items to their cart in the last 24 hours, but haven't received any other push notifications.

Diversion Point: ?

Control: No push notification.

Treatment: Personalized push notifications about item in the user cart.

Engagement Metrics: Push notifications open, mobile app visits, item views, orders completed, total orders placed, revenue.

Can we run multiple tests concurrently?

To answer this question, we will use Case studies 1 & 2. First we need to determine independence between each treatment group.

Independence: Does the welcome email affect whether or not we send push notifications?

Eligibility: We only want to include users that are eligibility. There may not be much overlap between these two groups.

Analysis: We want to consider what happens after the diversion point for each of these studies.

There are many things to consider when constructing this test. Are we sending an appropriate number of cases to the control and treatment groups? Are the events missing specific important data that we need, is that data delayed by real results?

2.4.3 SQL code for A/B testing

Listing 2.11: Applying an AB test on the test_assignments .

```
--How many test_assignments are there?
SELECT
  DISTINCT parameter_value
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
AND
  parameter_name = 'test_id'

--Check for potential problems with test assignments

SELECT
  parameter_value    AS test_id,
  date(event_time)   AS day,
  COUNT(*)           AS event_rows
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
GROUP BY
  parameter_value,
  date(event_time)
```

```
--Check for potential problems with test_id 5

SELECT
  user_id,
  COUNT(DISTINCT test_assignment) AS assignments
FROM
(
SELECT
  event_id,
  event_time,
  user_id,
  platform,
  MAX(CASE WHEN parameter_name = 'test_id'
    THEN CAST(parameter_value AS INT)
    ELSE NULL
      END) AS test_id,
  MAX(CASE WHEN parameter_name = 'test_assignment'
    THEN parameter_value
    ELSE NULL
      END) AS test_assignment
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
GROUP BY
  event_id,
  event_time,
  user_id,
  platform
ORDER BY
  event_id
) test_events
WHERE
  test_id = 5
GROUP BY
  user_id
ORDER BY
```

```
  assignments DESC

--Check for potential assignment problems.
--Make sure users are assigned to only one treatment group.

SELECT
  test_id,
  user_id,
  COUNT(DISTINCT test_assignment) AS assignments
FROM
(
SELECT
  event_id,
  event_time,
  user_id,
  platform,
  MAX(CASE WHEN parameter_name = 'test_id'
    THEN CAST(parameter_value AS INT)
    ELSE NULL
      END) AS test_id,
  MAX(CASE WHEN parameter_name = 'test_assignment'
    THEN parameter_value
    ELSE NULL
      END) AS test_assignment
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
GROUP BY
  event_id,
  event_time,
  user_id,
  platform
ORDER BY
  event_id
) test_events
GROUP BY
  test_id,
```

```
    user_id
ORDER BY
    assignments DESC
```

2.4.4 Create a new Metric

Start with a statistical hypothesis and then create a metric tied to a business value. Test Metrics allow us to answer specific questions with data, measure a products success, and to measure how a user behaviors are affected. Let's consider the following metric using the same case studies used before

New Metric: Measure a change resulting from a change in the user experience after the treatment.

Treatment events do not happen at the same time and not all users may be eligible to receive a treatment. Did users interact with the product you altered in any way? Many times, it might be simpler to use a binary metric but they might not cover enough knowledge. To recall, the important things to look at in the previous case studies were

Case Study 1: Push notifications opens, mobile app visits, item views.

Case Study 2: Orders completed, total orders placed, revenue.

Some of the tools we can use in our new metric are: Time boxes (just look at a period after exposure), Trimmed Metrics (Remove or cap the top percentile of samples from a noisy metric). Some code which shows how to do this with relevant questions is given in Listing 2.12.

Listing 2.12: Applying an AB test on the test_assignments .

```
--Using the table from Exercise 4.3, compute a metric that measures
--Whether a user created an order after their test assignment
--Requirements: Even if a user had zero orders, we should have a row that counts
-- their number of orders as zero
--If the user is not in the experiment they should not be included


SELECT
    test_events.test_id,
    test_events.test_assignment,
    test_events.user_id,
```

```
  MAX(CASE WHEN orders.created_at > test_events.event_time
  THEN 1
  ELSE 0
  END) AS orders_after_assignment
--This determines whether an order was made after the test event.
FROM
(
SELECT
  event_id,
  event_time,
  user_id,
  --platform,
  MAX(CASE WHEN parameter_name = 'test_id'
    THEN CAST(parameter_value AS INT)
    ELSE NULL
      END) AS test_id,
  MAX(CASE WHEN parameter_name = 'test_assignment'
    THEN parameter_value
    ELSE NULL
      END) AS test_assignment
-- MAX returns only one value between a NULL or test_assigment
-- When its NULL, it rejects that value.
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
GROUP BY
  event_id,
  event_time,
  user_id,
  platform
ORDER BY
  event_id
) test_events
LEFT JOIN
  dsv1069.orders
--By doing a left join, we are neglecting all users not in the experiment
ON
```

```
  orders.user_id=test_events.user_id
GROUP BY
  test_events.user_id,
  test_events.test_id,
  test_events.test_assignment

--Using the table from the previous exercise, add the following metrics
--1) the number of orders/invoices
--2) the number of items/line-items ordered
--3) the total revenue from the order after treatment

SELECT
  test_events.test_id,
  test_events.test_assignment,
  test_events.user_id,
  COUNT( DISTINCT(CASE WHEN orders.created_at > test_events.event_time
  THEN orders.invoice_id
  ELSE NULL
  END)) AS orders_after_assignment,
  COUNT( DISTINCT(CASE WHEN orders.created_at > test_events.event_time
  THEN orders.line_item_id
  ELSE NULL
  END)) AS items_after_assignment,
  SUM(CASE WHEN orders.created_at > test_events.event_time
  THEN orders.price
  ELSE 0
  END) AS total_revenue_after_treatment
FROM
(
SELECT
  event_id,
  event_time,
  user_id,
  MAX(CASE WHEN parameter_name = 'test_id'
    THEN CAST(parameter_value AS INT)
    ELSE NULL
      END) AS test_id,
  MAX(CASE WHEN parameter_name = 'test_assignment'
```

```
    THEN parameter_value
    ELSE NULL
      END) AS test_assignment
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
GROUP BY
  event_id,
  event_time,
  user_id,
  platform
ORDER BY
  event_id
) test_events
LEFT JOIN
  dsv1069.orders
ON
  orders.user_id=test_events.user_id
GROUP BY
  test_events.user_id,
  test_events.test_id,
  test_events.test_assignment
```

2.4.5 Analyzing Results

Here we will take the previous results and run them through in built AB testing tools through SQL. For the first metric we created it had the following properties:

Example: Order Binary.
Answers: How many users made an order?
Values: 1 or 0.
Average Value In [0,1].
Can Answer: Did the variant cause more users to place an order?

A mean metric might look something like:
Example: Number of Orders.
Answers: How many orders did a user make?

Values: Non-negative integers.
Average Value In [0,∞].
Can Answer: Did the variant cause users to create more orders?
The mean metric is presented in SQL in Listing 2.13

Listing 2.13: Applying an AB test for the average metric. The statistic from our results can be calculated on the website https://thumbtack.github.io/abba/demo/abba.html .

```
----Create the metric invoices (this is a mean metric, not a binary metric) and
----The count of users per treatment group for test_id=7
----The count of users with orders per treatment group.

SELECT
  user_level.test_assignment,
  COUNT(user_level.user_id) AS users,
  SUM(user_level.order_binary) AS users_with_orders
FROM
(
  SELECT
    assignments.test_id,
    assignments.test_assignment,
    assignments.user_id,
    MAX(CASE WHEN orders.created_at > assignments.event_time
    THEN 1
    else 0
    END) AS order_binary
  FROM
  (
  SELECT
    event_id,
    event_time,
    user_id,
    MAX(CASE WHEN parameter_name = 'test_id'
      THEN CAST(parameter_value AS INT)
      ELSE NULL
        END) AS test_id,
    MAX(CASE WHEN parameter_name = 'test_assignment'
      THEN parameter_value
      ELSE NULL
```

```
        END) AS test_assignment
  FROM
    dsv1069.events
  WHERE
    event_name = 'test_assignment'
  GROUP BY
    event_id,
    event_time,
    user_id,
    platform
  ORDER BY
    event_id
  ) assignments
  LEFT JOIN
    dsv1069.orders
  ON
    orders.user_id=assignments.user_id
  GROUP BY
    assignments.user_id,
    assignments.test_id,
    assignments.test_assignment
) user_level
WHERE
  user_level.test_id=7
GROUP BY
  user_level.test_assignment

--test_assignment zero is the control group
--test_assignment one is the treatment group

--The AB testing comes from a website
--https://thumbtack.github.io/abba/demo/abba.html

--Things that can go wrong with this analysis:
--There could be errors or bias introduced in the assignment process.
--The metric are not relevant to the hypothesis being tested.
--The metrics are not calculated properly.
--The statistics are not calculated properly.
```

```
--For the average metric, we will need statistics of
--an average number of users with orders,
--the standard deviation for this test statistic,
--and relevant p-values for given confidence intervals.

--Use the order_binary metric from the previous exercise,
--count the number of users per treatment group for test_id = 7,
--and count the number of users with orders (for test_id 7).

SELECT
   user_level.test_assignment,
  COUNT(user_level.user_id) AS users,
  SUM(user_level.order_binary) AS users_with_orders
FROM
(
  SELECT
     assignments.test_id,
     assignments.test_assignment,
     assignments.user_id,
    MAX(CASE WHEN orders.created_at > assignments.event_time
    THEN 1
     else 0
    END) AS order_binary
  FROM
  (
  SELECT
     event_id,
     event_time,
     user_id,
    MAX(CASE WHEN parameter_name = 'test_id'
      THEN CAST(parameter_value AS INT)
      ELSE NULL
        END) AS test_id,
    MAX(CASE WHEN parameter_name = 'test_assignment'
      THEN parameter_value
      ELSE NULL
        END) AS test_assignment
```

```
  FROM
    dsv1069.events
  WHERE
    event_name = 'test_assignment'
  GROUP BY
    event_id,
    event_time,
    user_id,
    platform
  ORDER BY
    event_id
  ) assignments
  LEFT JOIN
    dsv1069.orders
  ON
    orders.user_id=assignments.user_id
  GROUP BY
    assignments.user_id,
    assignments.test_id,
    assignments.test_assignment
) user_level
WHERE
  user_level.test_id=7
GROUP BY
  user_level.test_assignment

--Create a new item view binary metric.
--Count the number of users per treatment group,
--and count the number of users with views (for test_id 7)

SELECT
  user_level.test_assignment,
  COUNT(user_level.user_id) AS users,
  SUM(user_level.views_binary) AS users_with_views
FROM
(
  SELECT
    assignments.test_id,
```

```
  assignments.test_assignment,
  assignments.user_id,
  MAX(CASE WHEN views.event_time > assignments.event_time
  THEN 1
  else 0
  END) AS views_binary
FROM
(
SELECT
  event_id,
  event_time,
  user_id,
  MAX(CASE WHEN parameter_name = 'test_id'
    THEN CAST(parameter_value AS INT)
    ELSE NULL
      END) AS test_id,
  MAX(CASE WHEN parameter_name = 'test_assignment'
    THEN parameter_value
    ELSE NULL
      END) AS test_assignment
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
GROUP BY
  event_id,
  event_time,
  user_id,
  platform
ORDER BY
  event_id
) assignments
LEFT JOIN
(
SELECT
  *
FROM
  dsv1069.events
```

```
  WHERE
    event_name = 'view_item'
  ) views
  ON
    views.user_id=assignments.user_id
  GROUP BY
    assignments.user_id,
    assignments.test_id,
    assignments.test_assignment
) user_level
WHERE
  user_level.test_id=7
GROUP BY
  user_level.test_assignment

--Alter the result from before
--to compute the users who viewed an item WITHIN 30
--days of their treatment event

SELECT
  user_level.test_assignment,
  COUNT(user_level.user_id)          AS users,
  SUM(user_level.views_binary)       AS views_binary,
  SUM(user_level.views_binary_30d) AS views_binary_30d
FROM
(
  SELECT
    assignments.test_id,
    assignments.test_assignment,
    assignments.user_id,
    MAX(CASE WHEN views.event_time > assignments.event_time
    THEN 1
    else 0
    END) AS views_binary,
    MAX(CASE WHEN views.event_time > assignments.event_time
              AND DATE_PART('day',views.event_time-assignments.event_time) <= 30
        THEN 1
        else 0
```

```
    END) AS views_binary_30d

FROM
(
SELECT
  event_id,
  event_time,
  user_id,
  MAX(CASE WHEN parameter_name = 'test_id'
    THEN CAST(parameter_value AS INT)
    ELSE NULL
      END) AS test_id,
  MAX(CASE WHEN parameter_name = 'test_assignment'
    THEN parameter_value
    ELSE NULL
      END) AS test_assignment
FROM
  dsv1069.events
WHERE
  event_name = 'test_assignment'
GROUP BY
  event_id,
  event_time,
  user_id,
  platform
ORDER BY
  event_id
) assignments
LEFT JOIN
(
SELECT
  *
FROM
  dsv1069.events
WHERE
  event_name = 'view_item'
) views
ON
```

```
    views.user_id=assignments.user_id
  GROUP BY
    assignments.user_id,
    assignments.test_id,
    assignments.test_assignment
) user_level
WHERE
  user_level.test_id=7
GROUP BY
  user_level.test_assignment

--For the mean value metrics invoices, line_items, and total revenue compute the
--The count of users per treatment group
--The average value of the metric per treatment group
--The standard deviation of the metric per treatment group

SELECT
  mean_metrics.test_id,
  mean_metrics.test_assignment,
  COUNT(mean_metrics.user_id) AS users,
  AVG(mean_metrics.invoices)  AS exp_invoices,
  STDDEV(invoices)            AS stddev_inovices
FROM
(
SELECT
  assignments.test_id,
  assignments.test_assignment,
  assignments.user_id,
  COUNT( DISTINCT(CASE WHEN orders.created_at > assignments.event_time
  THEN orders.invoice_id
  ELSE NULL
  END)) AS invoices,
  COUNT( DISTINCT(CASE WHEN orders.created_at > assignments.event_time
  THEN orders.line_item_id
  ELSE NULL
  END)) AS line_items,
  SUM(CASE WHEN orders.created_at > assignments.event_time
  THEN orders.price
```

```
    ELSE 0
    END) AS total_revenue
FROM
(
SELECT
    event_id,
    event_time,
    user_id,
    MAX(CASE WHEN parameter_name = 'test_id'
      THEN CAST(parameter_value AS INT)
      ELSE NULL
        END) AS test_id,
    MAX(CASE WHEN parameter_name = 'test_assignment'
      THEN parameter_value
      ELSE NULL
        END) AS test_assignment
FROM
    dsv1069.events
WHERE
    event_name = 'test_assignment'
GROUP BY
    event_id,
    event_time,
    user_id,
    platform
ORDER BY
    event_id
) assignments
LEFT JOIN
    dsv1069.orders
ON
    orders.user_id=assignments.user_id
GROUP BY
    assignments.user_id,
    assignments.test_id,
    assignments.test_assignment
) mean_metrics
GROUP BY
```

```
  mean_metrics.test_id,
  mean_metrics.test_assignment
ORDER BY
  test_id

--If you wanted to, you could swap in for revenue or line_items
```