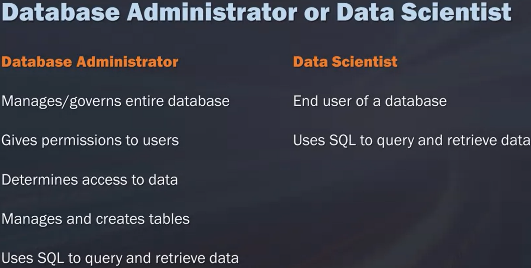
**SQL**

Fetch data and process it to derive valuable insights for decision making in organization

****

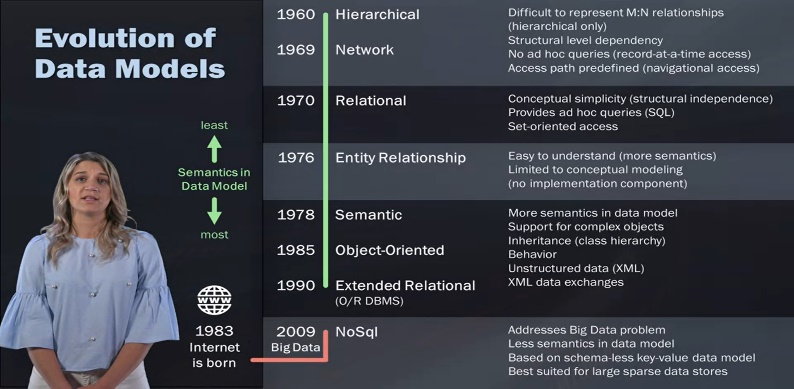
**Week 1**

**DB admin and Data scientist**

* Compare and contrast roles of DB admin and Data scientist
* Data scientist primarily use SQL for data retrieval, Combine and create tables, write queries for analysis and build test models

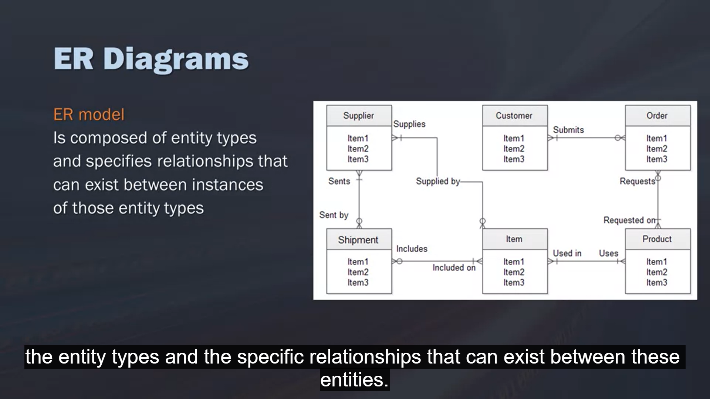
**SQL**

* SQL – Insert Query Update Modify [**C**reate **R**ead **U**pdate **D**elete : operations of database]
* Non procedural [cannot write applications]
* Descriptive statements to interact with database
* DBMS have dialect – SQL can translate

**Data models**

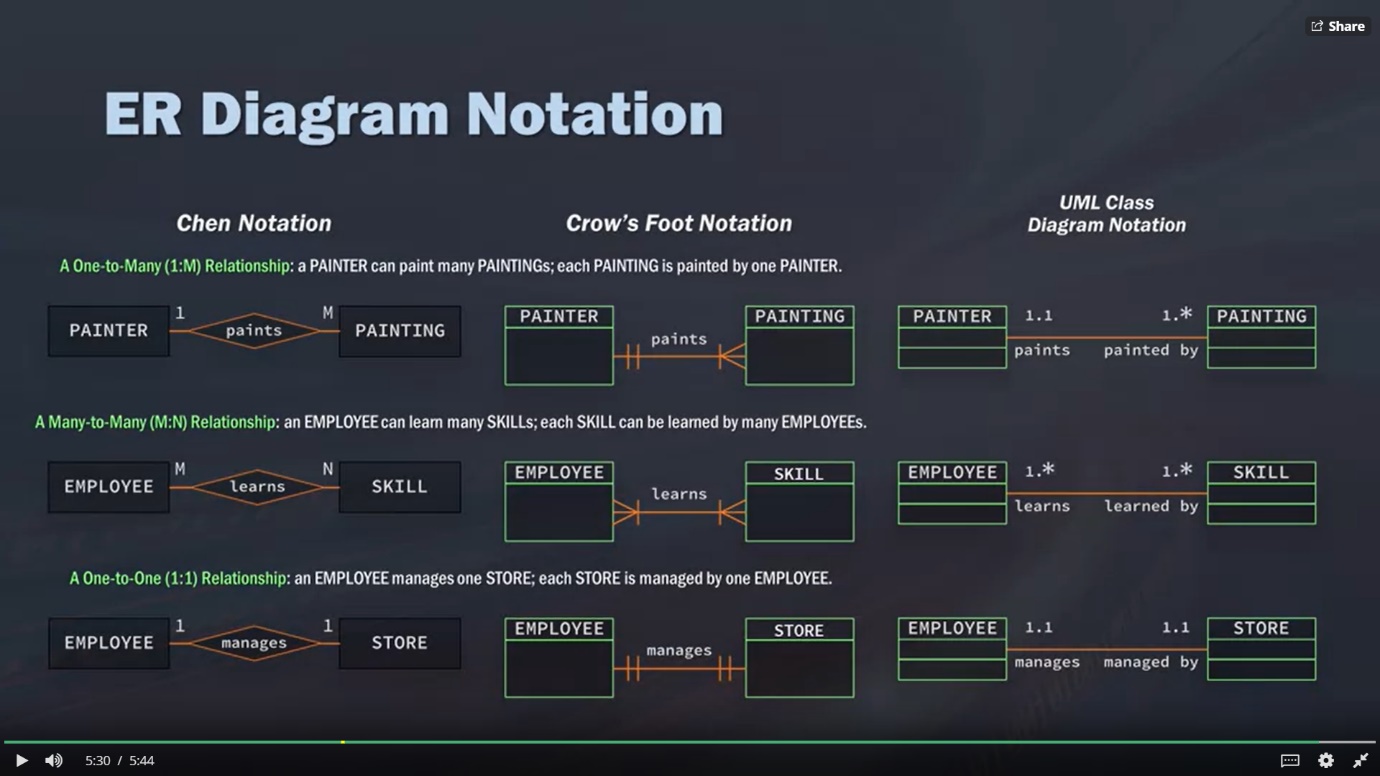
* Think about problem to solve before writing query
* Understand business process for which data is modeled, organized and structured
* Speeds up coding, less rework, improve accuracy
* RDB enables to write queries against structured data where relations between all tables are clearly defined

**Types of models**

* Relational model: Scalable and Structured [modeled] data for manipulation, analysis and logical querying using keys
* Transactional model: Operational data for transactions

**Modeling**

* Show relationships[one to many, many to many]
* Show links[Primary – uniquely ID rows & foreign keys – group of columns to ID rows]

**Notations in ER diagram**

**Reading material**

* [What is SQL and How is it Used?](https://www.thebalance.com/what-is-sql-and-uses-2071909)
* [NTC Hosting: Structured Query Language (it's worth exploring this site, not just this singular posting)](https://www.ntchosting.com/encyclopedia/databases/structured-query-language/)
* [SQLite Tutorial](https://www.tutorialspoint.com/sqlite/index.htm)
* [Norwalk Aberdeen: Entity-Relationship Diagrams (9 Minute YouTube Video)](https://www.youtube.com/watch?v=c0_9Y8QAstg)
* [Star Schema vs. Snowflake Schema](http://www.vertabelo.com/blog/technical-articles/data-warehouse-modeling-star-schema-vs-snowflake-schema)
* [Explain Star Schema & Snow Flake Design (5 Minute YouTube Video)](https://www.youtube.com/watch?v=KUwOcip7Zzc)
* [Data Modeling 101](http://www.agiledata.org/essays/dataModeling101.html)
* [What is Data Modeling - An Introduction for Business Analysts](http://business-analysis-excellence.com/what-is-data-modeling/)
* [Wikipedia: Data Modeling](https://en.wikipedia.org/wiki/Data_modeling)
* [Dataconomy: SQL vs. NoSQL - What You Need to Know](http://dataconomy.com/2014/07/sql-vs-nosql-need-know/)
* [TechRepublic: NoSQL keeps rising, but relational databases still dominate big data](http://www.techrepublic.com/article/nosql-keeps-rising-but-relational-databases-still-dominate-big-data/)
* [SiliconRepublic: Data Science Skills: Is NoSQL Better than SQL?](https://www.siliconrepublic.com/careers/data-science-skills-sql)

**Week 2**

**SQL Statements**

* **NULL : absence of everything**
* **Empty string : 0, spaces are present**
* **Comments :** block comment /\* \*/, Line comment - -

**Wildcard % \_**

* Used with LIKE for string parsing, Does not accept NULL
* All wildcards do not work with all DBs eg : \_ DB2, [] sqlite
* Helpful but reduces query performance; use operators as much as possible
* Add % sign in “optional” positions
* %pizza 🡪 ending with pizza
* %pizza% 🡪 has pizza as part of string
* [%@%.com](mailto:%25@%25.com) 🡪 match any email id
* Pi\_ \_ a 🡪 any 2 letters accepted between Pi and a

**Arithmetic operations () ^ / \* + -**

**SELECT a, (b-c)\*d as e FROM table1**

**Typical precedence**

* **SELECT** <column>
* **FROM** <table>
* **WHERE** <row level filter>
* **GROUP BY** <group by column>
* **HAVING** <condition on groups>
* **ORDER BY** <column with order>

|  |  |  |
| --- | --- | --- |
| **Statement** | **Syntax** | **Description** |
| **CRUD operations** | | |
| SELECT – FROM | SELECT <columns>  FROM <table>  LIMIT <first N records> |  |
| CREATE TABLE | CREATE TABLE < table > (  <col1> CHAR(10) PRIMARY KEY  <col2> INT(10) NOT NULL  .  .  ) |  |
| CREATE TEMPORARY TABLE | CREATE TEMPORARY TABLE <temp\_table> (  SELECT \*  FROM <table>  WHERE <col1> = [match value]  ) | Create fast temporary table, deleted after session terminated  simplify queries by creating a subset, and then joining to that subset, and derive a new calculation. |
| INSERT INTO | INSERT INTO < table >(  <col1>,  <col2>,  …  )  VALUES(  ‘val1’,  ‘Val2’,  …  ) |  |
| VIEW | CREATE VIEW <viewname> [IF NOT EXISTS]  AS  SELECT <col1>,<col2> ...  FROM <table>  WHERE <condition>; | Store part of data in memory for time being |
| UPDATE | UPDATE table <table>  SET <col1>= value1, <col2>= value2, ...  WHERE condition; |  |
| DELETE | DELETE FROM <table> WHERE condition; |  |
| **FILTER operations : DB are optimized for filtering**  Reduced amount of Specific[ queried ] data is pulled from database  Increased query performance and reduced load at client side | | |
| WHERE | SELECT <columns> FROM <table>  WHERE <col1> <operator> <match value> | **Filter on rows**  Operators  =, <>, >, <, >=, <=  IS NULL |
| BETWEEN | SELECT <columns> FROM <table>  WHERE <col1> BETWEEN <val1> AND <val2> | BETWEEN <LL Val> AND <UL Val > |
| IN | SELECT <columns> FROM <table>  WHERE <col1> IN (val1, val2,val3…) | IN (tuple of values and strings)  In is faster than OR  Sorted conditions are NOT required  Can include subqueries inside val |
| OR | SELECT <columns> FROM <table>  WHERE <col1> <operator> < value1> OR < value2> OR… | Order of operation is important |
| AND | SELECT <columns> FROM <table>  WHERE <col1> <operator> < value1> AND < value2> AND… | Use parenthesis around OR statement preceding AND to avoid “short circuiting” |
| NOT | SELECT <columns> FROM <table>  WHERE <col1> <operator> < value1> AND NOT < value2> AND… | Used to exclude a specific condition |
| LIKE | SELECT <columns> FROM <table>  WHERE <col1> LIKE “%abc%” | Wildcard for string parsing |
| ORDER BY | SELECT <columns> FROM <table>  ORDER BY <col1>, <col2>… [ASC|DESC] | Last clause on select statement  Can use a table column which is not SELECT as part of output  Can use column numbers  **ASC|DESC** works only on immediate preceding column |
| **Aggregate functions** | | |
| AVG | SELECT AVG(<column>) AS <avgCol>  FROM <table> | Ignore NULL while calculating AVG |
| COUNT | SELECT COUNT(<column>) AS <countCol>  FROM <table> | COUNT (\*) Counts all rows in FROM clause **including** **NULL** if any  COUNT (column) Counts all rows in FROM clause **ignoring** **NULL** if any |
| MIN | SELECT MIN(<column>) AS <minCol>  FROM <table> | Locate outliers  Ignore NULL while calculating MIN |
| MAX | SELECT MAX(<column>) AS <maxCol>  FROM <table> | Locate outliers  Ignore NULL while calculating MAX |
| SUM | SELECT SUM(<column>) AS <sumCol>  FROM <table> | Ignore NULL while calculating SUM |
| GROUP BY | SELECT <columns> FROM <table>  GROUP BY <col1>, <col2>.. | Can use a table column only in SELECT clause [except calculated aggregation]  Null are grouped together if present  Use of ORDER BY is recommended |
| HAVING | SELECT <columns> FROM <table>  GROUP BY <col1>, <col2>..  HAVING COUNT(\*) <operator> <match value> | **GROUP BY – HAVING for group wise filtering** |
| **Distinct function** |  |  |
| DISTINCT | SELECT COUNT(DISTINCT <column>)  AS <countDistCol>  FROM <table> | Without Distinct, SQL considers all data  And ignore duplicates |

**Reading material**

SQL for R

* [SQLDF Package](https://cran.r-project.org/web/packages/sqldf/index.html)
* [Documentation](https://cran.r-project.org/web/packages/sqldf/sqldf.pdf)
* [Examples](https://www.r-bloggers.com/manipulating-data-frames-using-sqldf-a-brief-overview/)

SQL for Spark

* [Overview](https://spark.apache.org/docs/latest/sql-programming-guide.html#overview)
* [Documentation](https://spark.apache.org/docs/latest/sql-programming-guide.html)

SQL with Hadoop

* [Hive Overview](https://hive.apache.org/)
* [Documentation](https://cwiki.apache.org/confluence/display/Hive/LanguageManual)

SQL for Python

* [Python-SQL Package Documentation](https://pypi.python.org/pypi/python-sql)

**Week 3**

**Subqueries [SELECT FROM WHERE IN (SELECT FROM …) ]**

* **Definition:** Query inside other query with additional filtering criteria; Work on single and multiple tables

**E.g. Know region of each customer who had order with freight more than 100**

SELECT CustomerID, Region FROM Customers

WHERE CustomerID in (

SELECT CustomerID FROM Orders

WHERE Freight > 100

);

* **Calculated fields**

**E.g. Total order placed by every Customer**

SELECT Customer\_name, States,

(

SELECT COUNT(\*) AS Orders FROM Orders

WHERE Orders.CustomerID = Customers.CustomerID

)

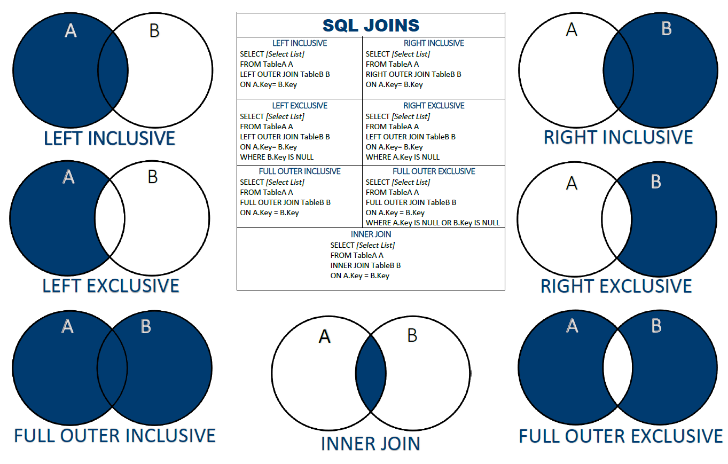
FROM Customers

ORDER BY Customer\_name;

* **Limitations:** Deeply nested subqueries reduce query performance; Subquery Selects only one column at a time

**Key Fields**

* Data Segmentation i.e. Breaking data into tables logically models Business problem
* It makes data Scalable, easier for manipulation and analysis, makes storage efficient
* Logical querying using Keys which indicate attributes through which tables are related

**Joins**

* Instead of duplicating tables; tabled can be **joined** using keys – a nonphysical single line query
* Associate related records from different tables on the fly

**Cross join :** It is Cartesian Join each row from T1 with each row of T2

* computationally taxing because if you have a table with just ten records in it and the second table with ten records, just performing a Cross Join is already going to increase it to 100.

**Inner Join:** matching value from both table [INTERSECTION]

* Computationally taxing to find ON conditions

**Left Join:** Values from left table with matching value from right table

**right Join:** Values from right table with matching value from left table

**Full outer Join:** Return all records where there is a match in *either* table one *or* there's a match in table two

**Union :** Combine results from multiple SELECT statements by **stacking results**

All SELECT columns must be in same sequence and have same data types

**Aliases:** table names short formed for better access

E.g. SELECT O.order\_ID FROM orders O;

**Pre-qualifiers** : subqueries for nested joins

E.g. SELECT P.product\_name, O.unit\_price, S.company

FROM (

( suppliers S INNER JOIN products P ON S.supplierID = P.supplierID )

INNER JOIN orders O ON P.productID = O.productID) ;

**Reading material**

* [Thinking in SQL vs Thinking in Python](https://blog.modeanalytics.com/learning-python-sql/)
* [Difference Between Union and Union All - Optimal Performance Comparison](https://blog.sqlauthority.com/2009/03/11/sql-server-difference-between-union-vs-union-all-optimal-performance-comparison/)

**Week4**

**Concatenating** : Link columns together **-** Use pipes

E.g. SELECT firstname, surname, firstname || surname FROM employees 🡪 firstname, surname, firstnamesurname

**Trimming** : TRIM everything from front and back OR RTRIM, LTRIM

E.g. SELECT TRIM (“ abc hdh “) AS trimmedString; 🡪 “abc hdh”

**Substring function** : select part of string - SUBSTR(<STRING>, A, B) gives substring starting at Ath character and return B subsequent characters

E.g. SELECT name, SUBSTR(“Jonathan”, 2,3) FROM employees 🡪 ona

**Change Case** - UPPER, LOWER, UCASE, LCASE

E.g. SELECT UPPER “myname” FROM employees 🡪 MYNAME

**Datetime function** : Different databases support different datatype : DATE, TIME, DATETIME, JULIANDAY, STRFTIME

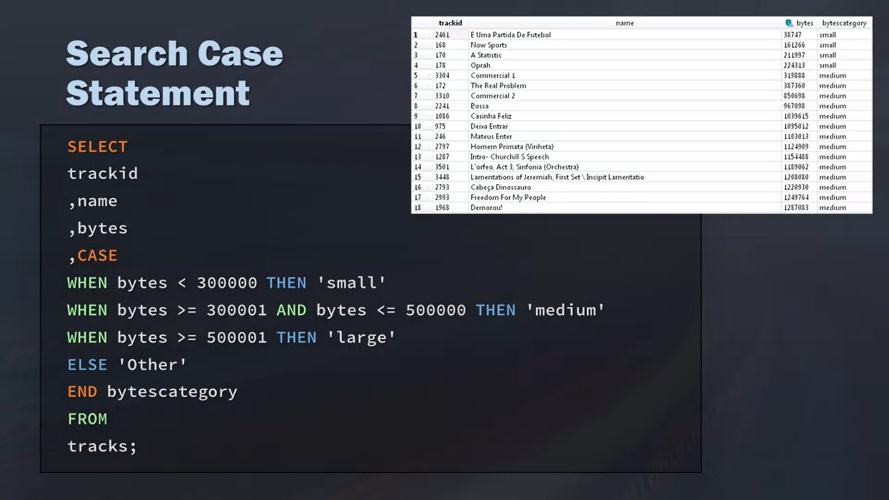
STRFTIME : extract certain elements of a date/time string

E.g.

* SELECT birthdate, STRFTIME(“%y”, birthdate) AS birthyear from birthdays
* SELECT DATE(‘now’) 🡪 return present system date
* SELECT birthdate, DATE((“now”) - birthdate) AS Present\_Age from birthdays 🡪 find current age from birth date

Time strings are extracted from DATETIME object

**Case statements** : Used inside SELECT, INSERT, DELETE, UPDATE to Categorize or Bin data

CASE WHEN THEN ELSE 🡪 IF THEN ELSE

**Views**

* A view is a stored query – an illusion of table
* Add remove columns without changing schema and database write limitations
* Tryout and encapsulate complex queries without ETL

SELECT \* FROM <myView>

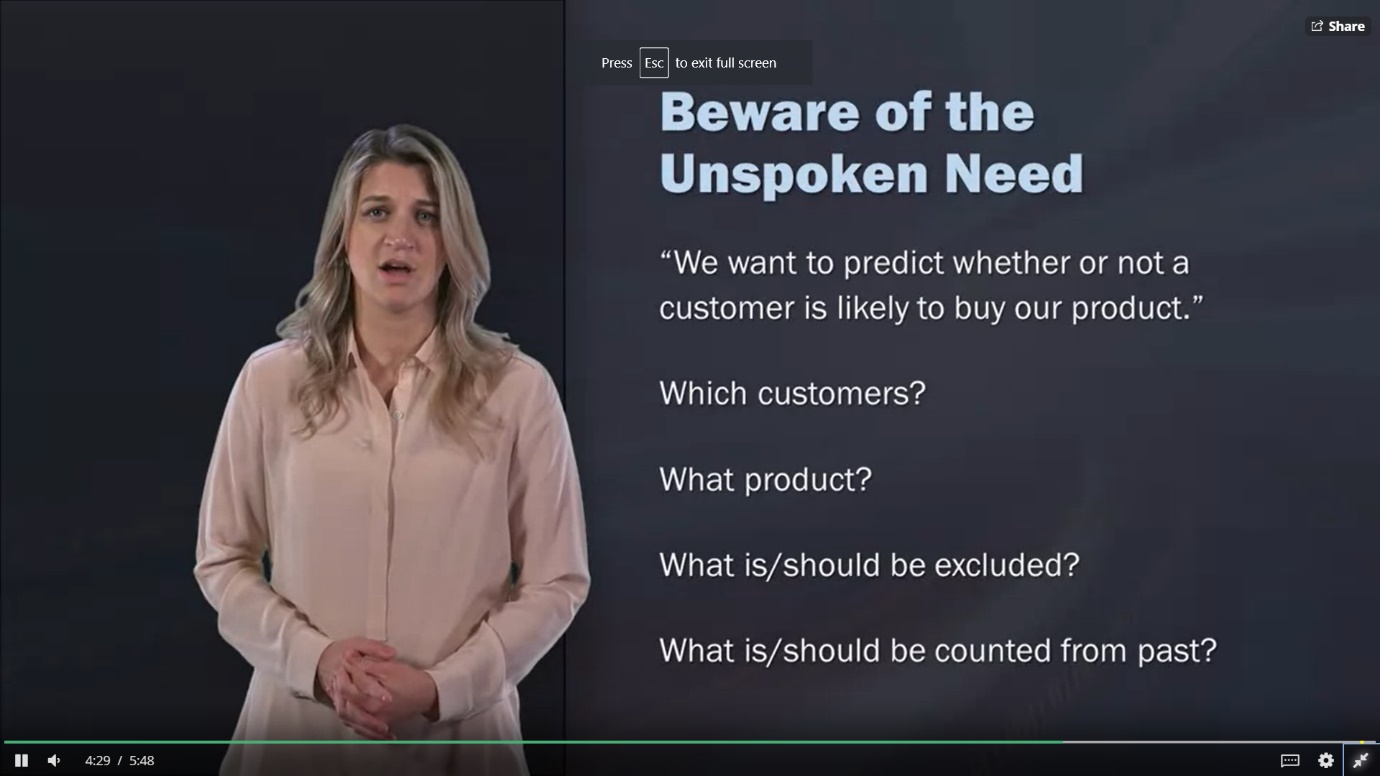
DROP VIEW <myView>

**Data Governance and Profiling**

Application of SQL in Data Science

**Data Governance**: looking at either descriptive statistics [columns MIN, MAX, DISTINCT, NULL, AVG] or different information on the data Rows [table size, last updated]

**Profiling** : Read and write policies about data site, clean up environment, escalation process



**Data understanding 🡨🡪 Business understanding**

* Unspoken requirements 🡪 requirement gathering

**While profiling data:**

* Details in data, create data model
  + draw out the different tables on a piece, of paper, basically creating my own data model and map.
* Consider joins and calculations necessary
* Data quality and format issues
* Test after each JOIN regressively
  + testing block by block allows you to find where any problems or issues occur
* Business rules – Date changes and Indicators
  + Has the data changed? Are the business rules different? Do you need to update and change the data indicators? Does anything need to be updated?

**Reading material**

* [SQL Authority: SQL Puzzles](https://blog.sqlauthority.com/category/sql-puzzle/)
* [SQLZOO](http://sqlzoo.net/)