Intro to SQL and Relational Databases

Core Reporting Concepts and High-Level Overview

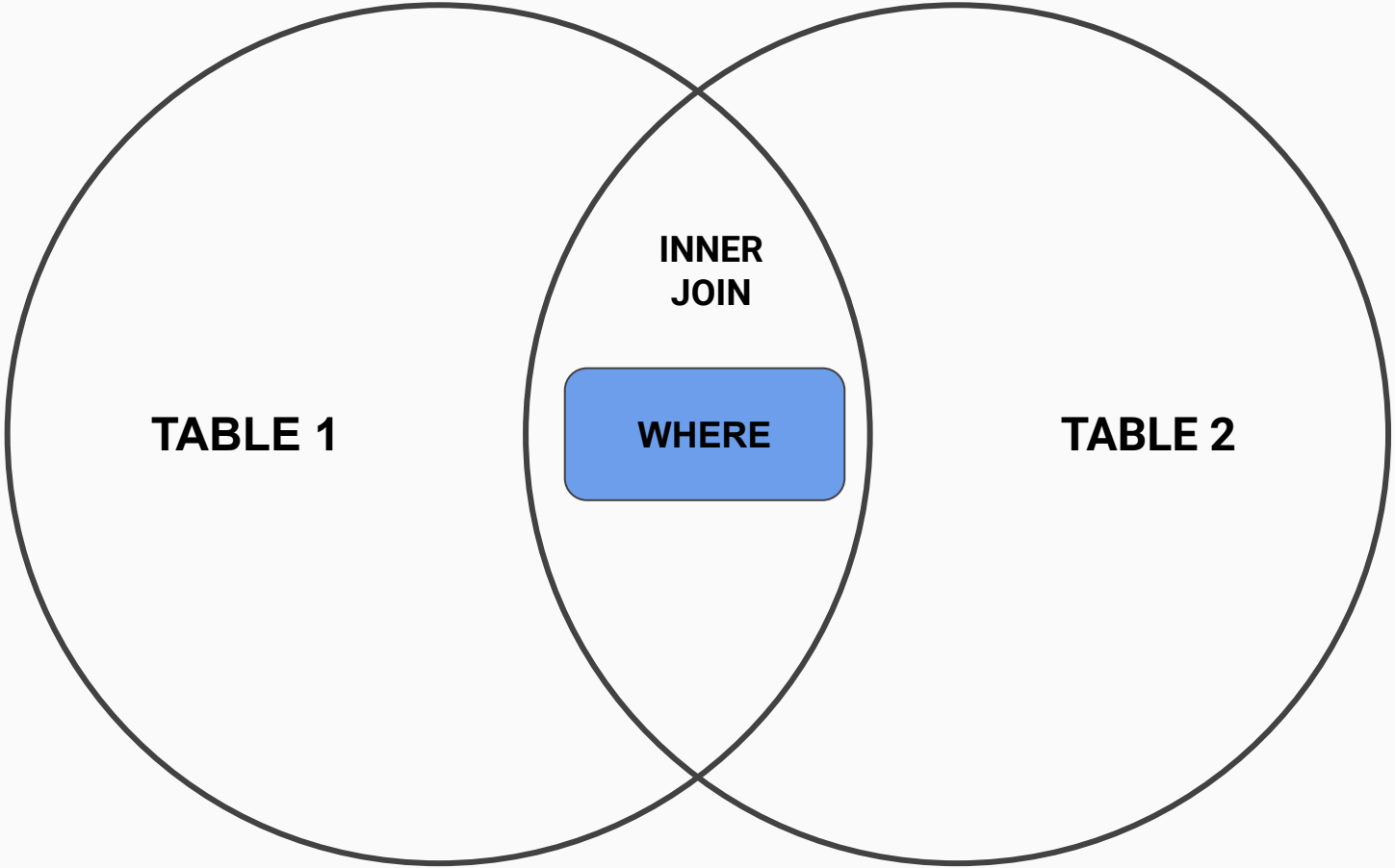
What SQL is NOT

- NOT an object-oriented language
- No need to define functions or write loops - tell it what you want and the database handles the rest
- No need for a compiler
- Unless you are performing Admin functions or updates, you are not typically telling SQL what to do or how to do it - instead, you are telling it what you want

What Does SQL Do?

- Combines and arranges data
- Relates multiple tables or data sets across common or key fields
- Creates segments or subsets of data based on shared or pre-defined values

Basically, you are creating a big Venn diagram with the conditions you set in your SQL query. You may already be familiar with some of the basic concepts if you’re a frequent user of Excel.


INNER
JOIN
TABLE 1
WHERE
TABLE 2

Helpful Resources

- https://www.w3schools.com/sql/ - Beginner level tutorials with an interactive sandbox
- https://www.sqltutorial.org/ - Intermediate level concepts (windowing functions)
- StackExchange - Helpful for specific issues, especially if you are using SQL as part of another product
- Database-Specific Sites - SQL has several variants (ex: Microsoft SQL Server, Oracle Database, PostgreSQL) and syntax/commands may be slightly different in each one

Relational Databases

- Tables = Circles in your Venn diagram
- More compact and efficient than putting everything in one table
- Divides data by purpose (ex: demographic information, contact information, transactions, survey responses)
- Tables relate to one another using key values (primary key, foreign key)
- “Flat” data can be warehoused and made into relational tables
- SQL and relational databases are used everywhere

Mathematical Concepts

- Abstract Algebra - how objects are defined and how they relate to each other; formal logic principals
- Set Theory - creating sets and using logical operators to separate or combine data (AND, OR, UNION, etc)

It is helpful to think of conditions mathematically and in terms of True / False (or True / Not True) rather than in the way you might describe them in English

Our Goal Today

- Get a sense of how to think in sets
- Take a first look at some of the more common SQL commands
- Learn how to frame a problem
- Explore some self-study resources
- Think about best practices

Don’t worry about memorizing any commands yet - that’s why we have Google and StackExchange

First Things First

First Things First

- Each flavor of SQL is slightly different - you might see != instead of <> to show inequalities for instance. The queries and commands shown today are for Microsoft SQL Server
    - In Enterprise environments, you'll likely encounter SQL Server or Oracle. Custom tools or applications will likely use something open source like MySQL, PostgreSQL, or SQLite
- Data comes in types and lengths. Typically, string data and dates require have single quotes around specific values in the query, whereas numerical and (usually) boolean values will not. Take a look at the field layout of the tables in your database, if that information is available.
    - It's not uncommon to run into tables where all of the data types are text/CHAR, especially in tables that import data from a file drop. Keep this in mind when doing date/numerical math.
- Know which fields contain primary foreign keys. Failing that, know which combinations of fields may indicate a unique record or might make a composite key. This will help you build your queries.
- Think about the cardinality of your data if you're joining tables - is it one-to-one? One-to-many? Many-to-many? Is there a junction table? These relationships will dictate how many records are in your query results
- Does your data have fields that are required? Are any of the fields nullable or not nullable?

NULL Values

- NULL values are not equal to zero
- NULL values are not **not** equal to zero
- Empty strings and NULL values are not the same thing (sometimes they may be treated as if they are, however)
- NULL values must be handled with IS and IS NOT rather than = and <>
- If your results seem off and you're not entirely sure why, there's a good chance that NULLs may be the culprit

Let’s Look At Some Queries

Query Basics, Part 1

- SELECT - FROM - WHERE
- SELECT *
- SELECT TOP
- Aliasing
- AND, OR, IN, NOT IN
- >, <, =, <>, IS, IS NOT
- *, %

Queries Pt 1

	tbl_SurveyRespondents		
ID	**Name**	**Country**	**LastTestDate**
0001	Glen Campbell	United States	8/1/2022 11:49 AM
0002	Jerry Reed	Portugal	
0003	Buck Owens	United States	
0004	Dwight Yoakam	Maldives	6/14/2021 12:30 PM

What conclusions can we make about this data so far?

- ID values have leading zeros - this means they are text values and not numbers if we're looking at raw data
- LastResponseDate does not always have a value - this means this field is nullable
- Creator of sample data has questionable taste in music

Queries Pt 1

Show me everything in the table:

```
SELECT * FROM tbl_SurveyRespondents
```

Show me the first 3 records:

```
SELECT TOP 3 * FROM tbl_SurveyRespondents
```

Show me the ID numbers for respondents in the United States:

```
SELECT * FROM tbl_SurveyRespondents
WHERE Country = 'United States'
```

ID	Name	Country	LastTestDate
0001	Glen Campbell	United States	8/1/2022 11:49 AM
0003	Buck Owens	United States	

Queries Pt 1

Aliasing - lets you rename a table and also makes it easier to specify which table you want your data to come from if two tables contain the same information. In many cases, it's better to specify columns rather than use SELECT * for all cases.

```
SELECT * FROM tbl_SurveyRespondents
WHERE Country = 'United States'
```

becomes:

```
SELECT r.ID,
    r.Name,
    r.Country,
    r.LastTestDate
FROM tbl_SurveyRespondents r
WHERE r.Country = 'United States'
```

Queries Pt 1

AND and OR operators

- AND is exclusive. {condition 1} AND {condition 2} must be true to return a record.
  - Unless {condition 1} AND {condition 2} are both true, the entire statement is not true.
- OR is inclusive. {condition 1} OR {condition 2} could be true to return a record.
  - {condition 1} could be false while {condition 2} is true and the whole OR statement would be true.
- Using casual language can be tricky. You could say “I want records where {condition 1} is true **and** I want records where {condition 2} is true”. This is actually an OR statement.
- You could also say “If {condition 1} is true **or** {condition 2} is true, then do NOT return the record”. This is actually an AND statement.
- Do your best to think in terms of whether a record meets ALL of the stated conditions in your WHERE statement - English is ambiguous, math is not.

Queries Pt 1

```sql
SELECT r.ID, r.Name, r.Country, r.LastTestDate
FROM tbl_SurveyRespondents r
WHERE r.Country = 'United States' OR r.Country = 'Portugal'
```

Three Results

```sql
SELECT r.ID, r.Name, r.Country, r.LastTestDate
FROM tbl_SurveyRespondents r
WHERE r.Country = 'United States' AND r.Country = 'Portugal'
```

Zero Results

```sql
SELECT r.ID, r.Name, r.Country, r.LastTestDate
FROM tbl_SurveyRespondents r
WHERE r.Country <> 'United States' AND r.Country <> 'Portugal'
```

One Result

```sql
SELECT r.ID, r.Name, r.Country, r.LastTestDate
FROM tbl_SurveyRespondents r
WHERE r.Country <> 'United States' OR r.Country <> 'Portugal'
```

Four Results

Queries Pt 1

Other WHERE statement operators:

- >, <, >=, <=, BETWEEN (put the lowest value on the left)

```
1  SELECT r.ID, r.Name, r.Country, r.LastTestDate
2  FROM tbl_SurveyRespondents r
3  WHERE r.LastTestDate BETWEEN '1/1/2022' AND '12/31/2022'
```

What About Those NULLs?

NULL values use IS or IS NOT rather than = or <>

```
1  SELECT r.ID, r.Name, r.Country, r.LastTestDate
2  FROM tbl_SurveyRespondents r
3  WHERE r.LastTestDate IS NOT NULL
```

ID	Name	Country	LastTestDate
0001	Glen Campbell	United States	8/1/2022 11:49 AM
0004	Dwight Yoakam	Maldives	6/14/2021 12:30 PM

Wildcards

What if I only kinda know what I want?

```
1  SELECT r.ID, r.Name, r.Country, r.LastTestDate
2  FROM tbl_SurveyRespondents r
3  WHERE r.Name LIKE 'Glen%'
4
```

ID	Name	Country	LastTestDate
0001	Glen Campbell	United States	8/1/2022 11:49 AM

% can be used to replace an unknown amount of characters anywhere in the string. Other wildcards for single characters or sets of characters exist, but these are highly dependent on SQL version and less commonly used. Also, you must use LIKE rather than = when using wildcards.

Query Basics, Part 2

- ORDER BY
- GROUP BY
- SUM, MIN, MAX
- HAVING

Queries Pt 2

ORDER BY defaults to ascending - however it's best to specify ASC or DESC explicitly

- NULL values are dealt with differently across platforms - some consider them to be smaller than non-nulls, some consider them to be larger

GROUP BY will aggregate results by the field you group on. For this reason it's best to group on qualitative fields/dimensions rather than quantitative metrics/measurements. Also, this will naturally yield fewer results that an ungrouped query - it can have a similar effect to a SELECT DISTINCT query.

GROUP BY is rarely used alone - it is almost always used with an aggregate function (SUM, MIN, MAX, COUNT) although these aggregate functions do not necessarily need a GROUP BY statement.

HAVING is similar to a WHERE clause, but it's applied after an aggregate function, rather than before

Be careful pulling in too many fields - if you're not grouping by that field and it's not an aggregate function, it may simply be the value for the latest record and not the one related to the MAX/MIN/etc

Queries Pt 2

	tbl_SurvResponses		
ID	**RespondentID**	**StarRating**	**SurveyDate**
1	0001	3	5/10/2021 10:27 AM
2	0004	1	6/14/2021 12:30 PM
3	0001	4	2/20/2022 11:00 AM
4	0001	3	8/1/2022 11:49 AM
5	0007	5	8/3/2022 2:20 PM
6	0006	5	8/11/2022 9:30 AM
7	0012	4	8/24/2022 4:45 PM
8	0006	3	9/3/2022 3:30 PM

Queries Pt 2

```
7   SELECT sr.RespondentID, MAX(sr.StarRating) AS MaxRating
8   FROM tbl_SurvResponses sr
9   WHERE sr.SurveyDate >= '1/1/2022'
10  GROUP BY sr.RespondentID
11  HAVING MAX(sr.StarRating) >= 3
12  ORDER BY sr.RespondentID ASC
```

RespondentID	MaxRating
0001	4
0006	5
0007	5
0012	4

Query Basics, Part 3

- INNER JOIN
- LEFT (OUTER) JOIN
- RIGHT (OUTER) JOIN
- FULL (OUTER) JOIN
- CROSS JOIN

JOINS

	tbl_SurveyRespondents		
ID	**Name**	**Country**	**LastResponseDate**
0001	Glen Campbell	United States	8/1/2022 11:49 AM
0002	Jerry Reed	Portugal	
0003	Buck Owens	United States	
0004	Dwight Yoakam	Maldives	6/14/2021 12:30 PM
0006	Lee Hazlewood	United States	9/3/2022 3:30 PM
0007	Gram Parsons	Canada	8/3/2022 2:20 PM
0008	Kris Kristofferson	United States	
0011	Ray Price	Mexico	
0012	George Jones	Lithuania	8/24/2022 4:45 PM

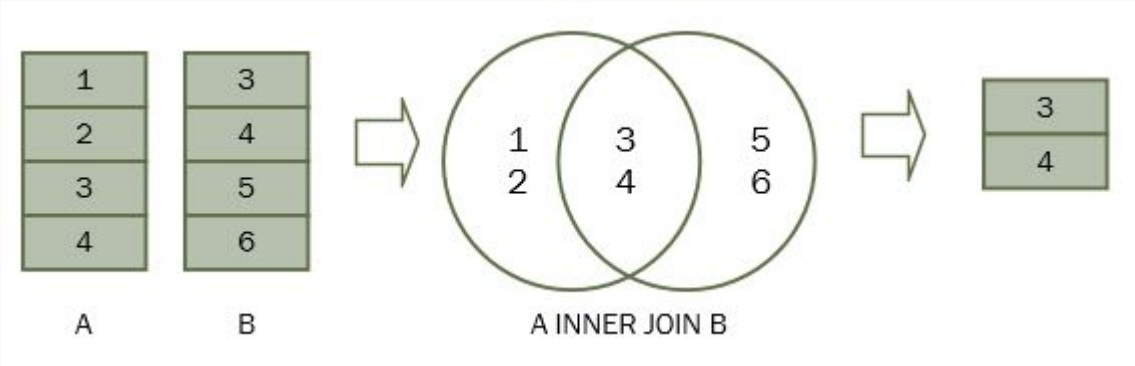
	tbl_SurvResponses		
ID	**RespondentID**	**StarRating**	**SurveyDate**
1	0001	5	5/10/2021 10:27 AM
2	0004	1	6/14/2021 12:30 PM
3	0001	4	2/20/2022 11:00 AM
4	0001	3	8/1/2022 11:49 AM
5	0007	5	8/3/2022 2:20 PM
6	0006	5	8/11/2022 9:30 AM
7	0012	4	8/24/2022 4:45 PM
8	0006	3	9/3/2022 3:30 PM

INNER JOIN

INNER JOINs return records where the field value in the left table is the same as the field value in the right table as specified in the ON statement. Records without a match are not returned. These records can include fields from either table, but be mindful of what you're selecting. If two tables have the same field, one may be more accurate than the other - use the proper alias to select the correct one.

Be careful of accidental cross joins. If you only want to return one record per ID number, consider SELECT DISTINCT or using an aggregate or windowing function if your tables don't have a one-to-one relationship.

In general, you can join as many tables as you want.



INNER JOIN

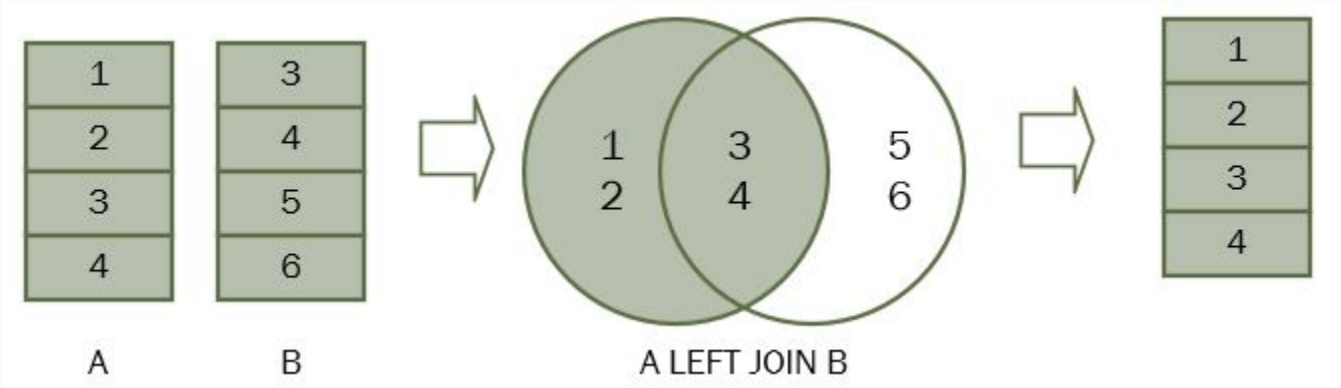
```
17  SELECT r.ID AS RespID, r.Name, sr.StarRating, sr.SurveyDate
18  FROM tbl_SurveyRespondents r
19  INNER JOIN tbl_SurvResponses sr
20  ON r.ID = sr.RespondentID
```

RespID	Name	StarRating	SurveyDate
0001	Glen Campbell	5	5/10/2021 10:27 AM
0004	Dwight Yoakam	1	6/14/2021 12:30 PM
0001	Glen Campbell	4	2/20/2022 11:00 AM
0001	Glen Campbell	3	8/1/2022 11:49 AM
0007	Gram Parsons	5	8/3/2022 2:20 PM
0006	Lee Hazlewood	5	8/11/2022 9:30 AM
0012	George Jones	4	8/24/2022 4:45 PM
0006	Lee Hazlewood	3	9/3/2022 3:30 PM

LEFT JOIN

LEFT JOINS return **all** records from the left table, plus matching records in the right table as specified by the ON statement. Values in the right table are considered to be NULL when there is no match for a record in the left table.

This is a highly useful join when combine with an IS NULL condition, especially for population segmentation or finding non-responders/non-engagers.



LEFT JOIN

```
17  SELECT r.ID AS RespID, r.Name, sr.StarRating, sr.SurveyDate
18  FROM tbl_SurveyRespondents r
19  LEFT JOIN tbl_SurvResponses sr
20  ON r.ID = sr.RespondentID
21  WHERE sr.RespondentID IS NULL
```

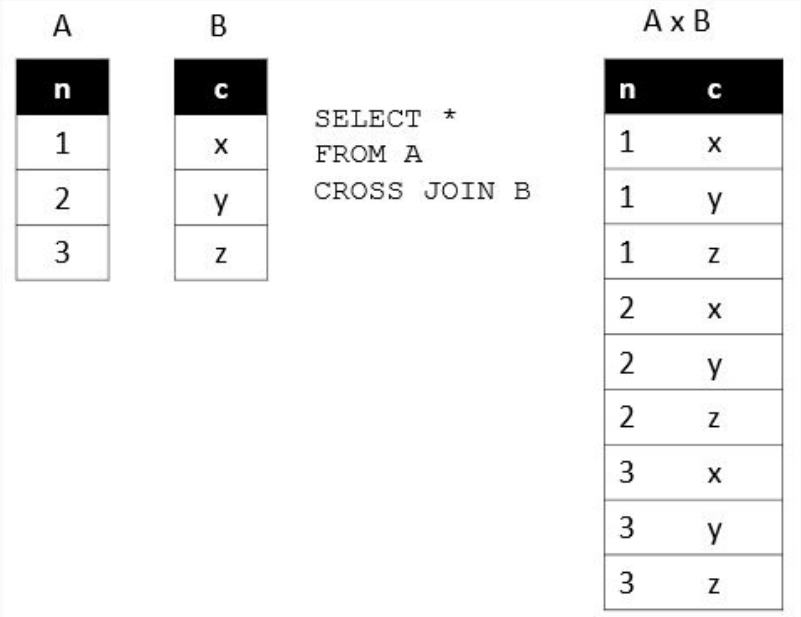
RespID	Name	StarRating	SurveyDate
0002	Jerry Reed		
0003	Buck Owens		
0008	Kris Kristofferson		
0011	Ray Price		

RIGHT JOIN / OUTER JOIN

Don’t worry about these ones

CROSS JOIN

CROSS JOINs create a Cartesian product - every record in the left table is matched with every record in the right table. As a result, there is no ON statement for this type of join. This is useful for creating sample lists or distribution lists.



Query Basics, Part 4

- COUNT
- DISTINCT

COUNT

COUNT is another aggregate function, like SUM, MIN, and MAX. As such, GROUP BY statements are not necessary but are frequently used in conjunction.

COUNT(*) operates differently than COUNT(fieldname), namely in that COUNT(*) includes NULL values in the count but COUNT(fieldname) does not.

COUNT can be used in conjunction with DISTINCT in order to tally up unique values

DISTINCT can be used in COUNT statements or as SELECT DISTINCT in order to remove non-unique results from your counts and queries. Much like with GROUP BY statements, be careful about pulling in too many fields (especially dates and other quantitative fields) as this can create uniqueness where you don't want it and cause duplicate records.

COUNT

```
25  SELECT COUNT(*) AS RepondentCount
26  FROM tbl_SurveyRespondents
```

RepondentCount
9

```
25  SELECT r.Country, COUNT(*) AS RepondentCount
26  FROM tbl_SurveyRespondents r
27  GROUP BY r.Country
28
```

Country	RepondentCount
Canada	1
Lithuania	1
Maldives	1
Mexico	1
Portugal	1
United States	4

```
25  SELECT COUNT(DISTINCT r.Country) AS CountryNumber
26  FROM tbl_SurveyRespondents r
27  GROUP BY r.Country
```

CountryNumber
6

Query Not-So-Basics

- Windowing functions (PARTITION BY / OVER / ROW_NUMBER)
- Subqueries
- Self-joins
- UNION
- DATEDIFF
- DATEPART
- CAST / CONVERT

Windowing Functions

Windowing functions allow you to partition your data by a group value and then sort or number records within each group. For instance, if we wanted to get the entire record associated with a minimum or maximum date and not just the min/max value itself, we'd use windowing functions (PARTITION / OVER) along with a ROW_NUMBER function.

```
30  SELECT ROW_NUMBER() OVER (PARTITION BY sr.RespondentID ORDER BY sr.SurveyDate DESC) AS row_num,
31      sr.RespondentID,
32      sr.SurveyDate
33  FROM tbl_SurvResponses sr
```

row_num	RespondentID	SurveyDate
1	0001	8/1/2022 11:49 AM
2	0001	2/20/2022 11:00 AM
3	0001	5/10/2021 10:27 AM
1	0004	2/20/2022 11:00 AM
1	0006	9/3/2022 3:30 PM
2	0006	8/11/2022 9:30 AM
1	0007	8/3/2022 2:20 PM
1	0012	8/24/2022 4:45 PM

Subqueries

Queries can be queried. Once you create a sub-query, it can be aliased and referenced just like a table. This is useful for when you arrange your data and want to select a subset (such as with windowing functions), for creating WHERE clause conditions (IN / NOT IN / EXISTS / NOT EXISTS), or for UNION functions.

```
29  SELECT sub.RespondentID, sub.SurveyDate FROM
30      (SELECT ROW_NUMBER() OVER (PARTITION BY sr.RespondentID ORDER BY sr.SurveyDate DESC) AS row_num,
31          sr.RespondentID,
32          sr.SurveyDate
33      FROM tbl_SurvResponses sr) sub
34  WHERE sub.row_num = 1
```

RespondentID	SurveyDate
1	8/1/2022 11:49 AM
4	2/20/2022 11:00 AM
6	9/3/2022 3:30 PM
7	8/3/2022 2:20 PM
12	8/24/2022 4:45 PM

Self-Joins

Sometimes instead of comparing one table to another table, we want to compare records to other records within the same table. This can be easily done by giving two aliases to the same table and joining them together. A frequently-referenced example of this is figuring out employee hierarchy from a single table.

Id	FullName	Salary	ManagerId
1	John Smith	10000	3
2	Jane Anderson	12000	3
3	Tom Lanon	15000	4
4	Anne Connor	20000	
5	Jeremy York	9000	1

```
SELECT
    employee.Id,
        employee.FullName,
        employee.ManagerId,
        manager.FullName as ManagerName
FROM Employees employee
JOIN Employees manager
ON employee.ManagerId = manager.Id
```

Id	FullName	ManagerId	ManagerName
1	John Smith	3	Tom Lanon
2	Jane Anderson	3	Tom Lanon
3	Tom Lanon	4	Anne Connor
5	Jeremy York	1	John Smith

UNION

Multiple subqueries can be joined together as long as the output field layouts are the same. UNION will remove duplicate records, whereas UNION ALL will leave duplicates intact. This may differ depending on what kind of SQL you are using, and SELECT * statements may cause unwanted uniqueness if field layouts from the source tables are different.

```
38  SELECT jan.ID,jan.Name, jan.Country, jan.LastTestDate
39  FROM tbl_SurveyRespondents_January2022 jan
40
41  UNION
42
43  SELECT feb.ID, feb.Name, feb.Country, feb.LastTestDate
44  FROM tbl_SurveyRespondents_February2022 feb
```

DATEDIFF / DATEPART

Often times you'll be wanting to run the same query on a regular basis using a rolling date range (ex: the previous week/month relative to the time you are running it). This can be done using a DATEDIFF function. The query below selects records from the previous day from tbl_SurvResponses:

```
48  SELECT sr.ID, sr.RespondentID, sr.StarRating, sr.SurveyDate
49  FROM tbl_SurvResponses sr
50  WHERE DATEDIFF(day, sr.SurveyDate, GETDATE()) = 1
```

The DATEDIFF function has three arguments: interval, date1, and date2. In the example above, the interval is "day", date1 is the survey date, and date2 is the current date and time. This DATEDIFF condition is true when there is an interval of exactly one day between SurveyDate time and the time that the query is run.

Both DATEDIFF and the DATEPART functions will ONLY look at the interval value (or difference in interval value) between the two dates - it does not actually calculate the length of time between them. For instance, for an interval of "day", the interval value between 8/1/2022 11:59pm and 8/2/2022 12:01am is 1. For an interval of "month", the difference between 8/1/2022 and 9/1/2022 is 1, and the difference between 8/31/2022 and 9/1/2022 is also 1.

CAST / CONVERT

Sometimes data is not in the format that you need in order to do date or number calculations. Most often, this is because a table is set up such that the data type for every field is text/CHAR/VARCHAR, since that is technically how most flat files and CSVs are delivered. Alternately, you may have received a date in the wrong format and you want to convert it before exporting it to a file. You can fix these issues by using CAST or CONVERT. For example, if a number field is stored as text, you may want to use something like:

```
53    SELECT CAST(AvgResponseTime AS decimal) AS AvgResponse
```

If a date is stored as datetime and you want to convert it to text to remove the timestamp before exporting it, you may want to use:

```
52
53    SELECT CONVERT(varchar, ResponseDate, 101) AS RespDate
```

These two functions behave largely the same, except CONVERT offers more flexibility for formatting the date (the 101 in the example above indicates a standard US format with no timestamp)

Guidelines and Gotchas

Gs and Gs

- The most important thing is to frame your problem correctly. 90 percent of everything else can be cobbled together from StackExchange and Google.
- Get in the habit of using aliases for tables.
- Use in-line comments (/* */ or –) if the platform allows. There's a good chance that you'll forget what you did and why, so code as if someone else will be reading your query.
- Take the time to wrap your head around the way that operators work in a mathematical sense. "AND" conditions are more exclusive, "OR" conditions are more inclusive - this can be counterintuitive and even seem backwards if you're trying to describe your problems in colloquial English. Think about what must be trust and what must not be true.
- NULLs will jam you up. If you've got an issue you can't figure out or if you seem to be missing records, there's a good chance that NULLs are the problem.
- Data type mismatches will jam you up. It's very common to run into cases where an entire table is imported as text fields, because field metadata (type and length) can't be communicated in flat files. To do arithmetic or date functions, you MUST use the correct data types.
    - Similarly, make sure that decimal fields are not being stored/treated as integers
- Consider the source of your data. Never assume that it is clean or formatted correctly until you've had a chance to work with it, ESPECIALLY if there are fields sourced from user input
- If queries take a long time to run, consider splitting them into multiple steps or doing things to optimize them. Don't do too much in a single query, and use JOINs instead of IN / NOT IN.

Steal Liberally

- Don’t reinvent the wheel
- Don’t waste time solving problems that have already been solved
- Keep an archive of any query or code you write that you use regularly or solves something in a unique way (ambiguate if necessary)

Uh Oh, Something Went Wrong

Troubleshooting Scenarios

- Identifying and fixing problems is more important than being able to write queries off the top of your head. You can Google commands and functions but you can’t Google common sense.
- Know your data. Make sure field types for primary/foreign keys match, make sure you know what can and cannot be null.
- Keep stuff as simple as possible.

I Have Too Many Records!

- Check your WHERE statement. Did you use an OR instead of an AND? Did you include enough constraints?
- Check your JOINs. Did you join on the right field? Did you accidentally create a Cartesian product (cross join)? Were you expecting one-to-one cardinality in your tables, and if so does the data match up with this?
    - SELECT / COUNT DISTINCT on a key field - does the result count match the record count in the table?

I Don’t Have Enough Records!

- Check your WHERE statement. Did you use “AND” instead of “OR”? An = instead of a LIKE? Did you properly account for NULLs? Are your date ranges correct (i.e. did you use >= and <= where appropriate rather than > and <)? Did a DATE get imported as DATETIME with a midnight time stamp?
- Check your JOINs. Did you join on an empty field? Is your data actually related?
- Did you use a NOT IN condition subquery on a field with a null value in it?

I Can’t Get This Dang Thing To Run!

- Are you doing too much in one query? Split it up into simpler and more digestible steps. Add conditions line by line until you find the problem.
- Do you need to use a self-join or a subquery? Tables can be referenced multiple times in one query, and subqueries themselves can be treated as tables and even aliased.
- If you are inserting data into a table, are you making a primary key violation?
- Did you add a comma between your last field name and the FROM statement on accident?
- Debugging is not always easy depending on the environment you are in, so take it line by line and be patient.