

QSS20: Modern Statistical Computing

Unit 07: SQL

Outline

- ▶ **SQL: ways of interacting with a database and starting connection**
- ▶ Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- ▶ Subqueries, aggregations, and joins: one table
- ▶ Subqueries, aggregations and joins: two tables

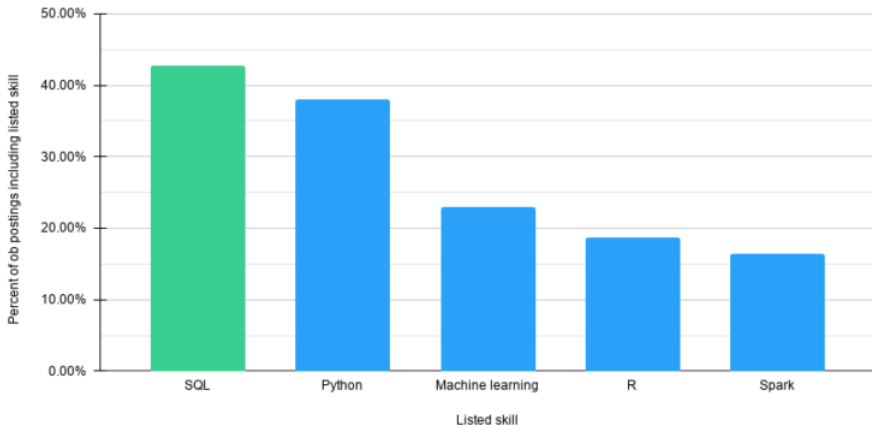
What is SQL and why might it be useful?

- ▶ **StructuredQueryLanguage**
- ▶ While relatively uncommon in academia, many companies / governments expect data scientists to be able to write SQL queries
- ▶ In turn, a particular data warehouse/database might use different varieties of database engines to store data: Amazon Redshift; MySQL; postgresSQL; Microsoft SQL server; SQLite
- ▶ Nearly identical syntax but some small differences on the margins; here, we're using a MySQL database since it's what Dartmouth Research Computing hosts!

What is SQL and why might it be useful?

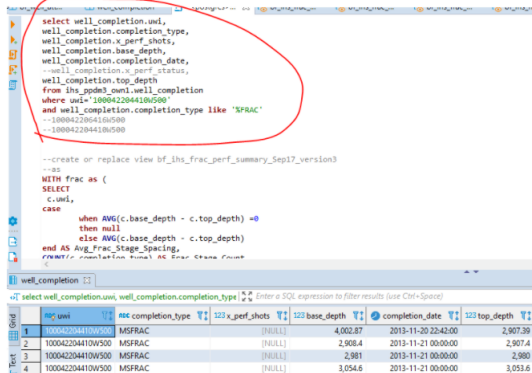
Percent of All Data Jobs Listing SQL

Data Source: Indeed.com, 1/29/2021



One way of writing SQL queries / viewing parts of a database

IDEs that are similar to RStudio, pycharm, or Jupyter notebooks that allow you to preview tables in a database and write/execute queries in a console or via a .sql script:



The screenshot shows a database IDE interface. The top pane contains a SQL script. A red circle highlights the first part of the script, which is a SELECT query. Below the script, the results of the query are displayed in a table. The table has 7 columns: well_completion.wei, well_completion.completion_type, well_completion.x_perf_shots, well_completion.base_depth, well_completion.completion_date, well_completion.x_perf_status, and well_completion.top_depth. The results show 4 rows of data.

```
select well_completion.wei,  
well_completion.completion_type,  
well_completion.x_perf_shots,  
well_completion.base_depth,  
well_completion.completion_date,  
--well_completion.x_perf_status,  
well_completion.top_depth  
from ihs_opdes_0wnl.well_completion  
where wei='100042204410W500'  
and well_completion.completion_type like '%FRAC'  
--100042206410W500  
--100042204410W500  
  
--Create or replace view bf_ihs_frac_perf_summary_Sep17_version3  
--as  
WITH frac as (  
SELECT  
c.wei,  
case  
when AVG(c.base_depth - c.top_depth) =0  
then null  
else AVG(c.base_depth - c.top_depth)  
end AS Avg_Frac_Stage_Spacing,  
COUNT(c.completion_type) AS Frac_Stage_Count  
)  
select well_completion.wei,  
well_completion.completion_type,  
well_completion.x_perf_shots,  
well_completion.base_depth,  
well_completion.completion_date,  
well_completion.x_perf_status,  
well_completion.top_depth  
from ihs_opdes_0wnl.well_completion  
where wei='100042204410W500'  
and well_completion.completion_type like '%FRAC'  
--100042206410W500  
--100042204410W500  
  
--Create or replace view bf_ihs_frac_perf_summary_Sep17_version3  
--as  
WITH frac as (  
SELECT  
c.wei,  
case  
when AVG(c.base_depth - c.top_depth) =0  
then null  
else AVG(c.base_depth - c.top_depth)  
end AS Avg_Frac_Stage_Spacing,  
COUNT(c.completion_type) AS Frac_Stage_Count  
)  
select well_completion.wei,  
well_completion.completion_type,  
well_completion.x_perf_shots,  
well_completion.base_depth,  
well_completion.completion_date,  
well_completion.x_perf_status,  
well_completion.top_depth  
from ihs_opdes_0wnl.well_completion  
where wei='100042204410W500'  
and well_completion.completion_type like '%FRAC'  
--100042206410W500  
--100042204410W500
```

	well_completion.wei	well_completion.completion_type	well_completion.x_perf_shots	well_completion.base_depth	well_completion.completion_date	well_completion.x_perf_status	well_completion.top_depth
1	100042204410W500	MSFRAC	[NULL]	4,002.87	2013-11-20 22:42:00		2,907.39
2	100042204410W500	MSFRAC	[NULL]	2,908.4	2013-11-21 00:00:00		2,907.4
3	100042204410W500	MSFRAC	[NULL]	2,981	2013-11-21 00:00:00		2,980
4	100042204410W500	MSFRAC	[NULL]	3,054.6	2013-11-21 00:00:00		3,053.6

Source: StackOverflow

Another way of interacting with database: connecting via another scripting language and sending queries through the connection

1. Use an R or Python package that helps you connect with a specific type of database (Python: SQLAlchemy; MySQL connector; pyodbc; etc.; similar ones in R)
2. Establish a connection between your local computer and the database
3. Write a SQL query
4. Execute the query
5. Pull the result and work with the result in that language

Preliminary step: load credentials and establish a connection

```
1 ## import mysql connector
2 import mysql.connector
3
4 ## load creds
5 creds = load_creds("../private_data/creds_forclass.yaml")
6
7 ## use username, pwd, host, port, etc
8 ## to establish a connection to the database
9 cnx = mysql.connector.connect(
10     user=creds['practice_database']['db_user'],
11     password=creds['practice_database']['db_password'],
12     port=creds['practice_database']['port'],
13     database= creds['practice_database']['database'],
14     host = creds['practice_database']['host'])
```

Working example: two tables from Chicago felony prosecution datasets used in pset 1

Desc.	Table	Main cols	Database
Initiations	caseinit	CASE_ID; CASE_PARTICIPANT_ID; RACE; GENDER; UPDA- TED_OFFENSE_CATEGORY; is_in_diversion	rjohnson
Diversions	divert	CASE_ID; CASE_PARTICIPANT_ID; RACE; DIVERSION_PROGRAM; OF- FENSE_CATEGORY	rjohnson

Outline

- ▶ SQL: ways of interacting with a database and starting connection
- ▶ **Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions**
- ▶ Subqueries, aggregations, and joins: one table
- ▶ Subqueries, aggregations and joins: two tables

Basic syntax of a SQL query

- ▶ Select **specific** columns and rows that meet condition:

```
select col1, col2  
from tablename  
where somecondition holds
```

- ▶ Select **all** columns and rows that meet condition:

```
select *  
from tablename  
where somecondition holds
```

Examining structure of data: selecting first 10 rows from case initiations table

```
1 ## define a query
2 sample_case_q = """
3 select *
4 from caseinit
5 limit 10
6 """
7 ## feed read sql query the query and my db connection
8 read_sample_d = pd.read_sql_query(sample_case_q, cnx)
```

Breaking things down:

- ▶ `select *`: select all columns
- ▶ `from caseinit`: which table in database to pull from (if our database was more complicated, might be structured as something like `sentencing_schema.caseinit` that would indicate the case initiations table in the sentencing schema)
- ▶ Feed the (1) query and (2) database connection to pandas `read_sql_query`

Columns available to select from

```
Index(['CASE_ID', 'CASE_PARTICIPANT_ID', 'RECEIVED_DATE', 'OFFENSE_CATEGORY',  
      'PRIMARY_CHARGE_FLAG', 'CHARGE_ID', 'CHARGE_VERSION_ID',  
      'CHARGE_OFFENSE_TITLE', 'CHARGE_COUNT', 'CHAPTER', 'ACT', 'SECTION',  
      'CLASS', 'AOIC', 'EVENT', 'EVENT_DATE', 'FINDING_NO_PROBABLE_CAUSE',  
      'ARRAIGNMENT_DATE', 'BOND_DATE_INITIAL', 'BOND_DATE_CURRENT',  
      'BOND_TYPE_INITIAL', 'BOND_TYPE_CURRENT', 'BOND_AMOUNT_INITIAL',  
      'BOND_AMOUNT_CURRENT', 'BOND_ELECTRONIC_MONITOR_FLAG_INITIAL',  
      'BOND_ELECTROINIC_MONITOR_FLAG_CURRENT', 'AGE_AT_INCIDENT', 'RACE',  
      'GENDER', 'INCIDENT_CITY', 'INCIDENT_BEGIN_DATE', 'INCIDENT_END_DATE',  
      'LAW_ENFORCEMENT_AGENCY', 'LAW_ENFORCEMENT_UNIT', 'ARREST_DATE',  
      'FELONY_REVIEW_DATE', 'FELONY_REVIEW_RESULT',  
      'UPDATED_OFFENSE_CATEGORY', 'is_in_diversion'],  
      dtype='object')
```

Columns: selecting specific columns with no transformations/additions

```
1 select CASE_ID, CASE_PARTICIPANT_ID  
2 from caseinit
```

What this does: selects those case and participant identifier from the case initiations table

Rows: filtering to specific rows using where

```
1 select CASE_ID, CASE_PARTICIPANT_ID,  
2 AGE_AT_INCIDENT  
3 from caseinit  
4 where AGE_AT_INCIDENT > 40
```

Other logical operations:

- ▶ Equals: =
- ▶ Not equals: <>

Rows: filtering to specific rows using in or like

► Specify categories:

```
1 select CASE_ID, CASE_PARTICIPANT_ID,  
2 RACE  
3 from caseinit  
4 where RACE in ("Black", "HISPANIC")
```

► If contains Black anywhere in RACE string

```
1 select CASE_ID, CASE_PARTICIPANT_ID,  
2 RACE  
3 from caseinit  
4 where RACE like '%Black%'
```

Columns: creating new columns based on conditions

CASE, WHEN, ELSE syntax works similar to `np.where` and `np.select`

```
1 select *,
2 CASE
3     WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
4     THEN 'Same offense'
5     ELSE 'Diff offense'
6 END charge_update
7 from caseinit
```


What if we want to create a new col and then filter using that same columns as part of the same query? Query

If we try this query (created the charge_update column and then row filtering):

```
1 select *,
2 CASE
3     WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
4     THEN 'Same offense '
5     ELSE 'Diff offense '
6 END charge_update
7 from caseinit
8 where charge_update = 'Diff offense '
```

What if we want to create a new col and then filter using that same columns as part of the same query? Error

Get this SQL code error where it's telling us that it doesn't recognize the new column, because we can't simultaneously create a new col and filter:

```
DatabaseError: Execution failed on sql '
select *,
CASE
    WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY THEN 'Same offense'
    ELSE 'Diff offense'
END charge_update
from caseinit
where charge_update = 'Diff offense'
': 1054 (42S22): Unknown column 'charge_update' in 'where clause'
```

Approach one: direct row filtering using where without the case when

```
1 select *  
2 from caseinit  
3 where OFFENSE_CATEGORY <> UPDATED_OFFENSE_CATEGORY
```

Outline

- ▶ SQL: ways of interacting with a database and starting connection
- ▶ Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- ▶ **Subqueries, aggregations, and joins: one table**
- ▶ Subqueries, aggregations and joins: two tables

Approach using subqueries: in words

1. Write a **subquery** to create the column indicating whether the charge has been updated (`charge_update`)
2. Use the output of that subquery
3. Then, in the main select column, we can select/do whatever we want with the `charge_update` column we created in the subquery

Approach using subqueries: in code

```
1 select *
2 from caseinit
3 inner join
4     (select CASE_ID as cid ,
5      CASE_PARTICIPANT_ID as cpid ,
6      CASE
7          WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
8          THEN 'Same offense '
9          ELSE 'Diff offense '
10      END charge_update
11     from caseinit) as tmp
12 on tmp.cid = caseinit.case_ID and
13 tmp.cpid = caseinit.CASE_PARTICIPANT_ID
14
15 where charge_update = "Diff offense"
```

Breaking things down, we use the parentheses to define a **subquery** where we:

- ▶ Use “as” to alias CASE_ID as cid, similar with cpid
- ▶ Execute our case when statement
- ▶ Alias the newly created table as tmp and join back w/ our main data

Subqueries are most powerful in the context of aggregations

General workflow:

1. Construct a **subquery** that does some aggregation of the table
2. Join the result of that aggregation to the main table
3. Do operations like row and column filtering in the outer part of the query that uses the output of the subquery

Example: disparities in who receives leniency through diversion

Want to:

1. Find the five most common offenses in the `caseinit` table
2. For those five most common offenses, find the percent of Black defendants whose cases are diverted and the percent of White defendants whose cases are diverted
3. Create a new column—`diff_diversion`—that's the White diversion rate for the offense minus the Black diversion rate

Rather than creating a complex query all at once, let's incrementally build the query

Step 1: finding five most common offenses

```
1 select UPDATED_OFFENSE_CATEGORY,  
2 count(*) as count_offense  
3 from caseinit  
4 where RACE in ("Black", "White")  
5 group by UPDATED_OFFENSE_CATEGORY  
6 order by count_offense desc  
7 limit 5
```

Breaking it down:

- ▶ Grouping by offense category
- ▶ Using `count(*)` to get the number of rows in that group
- ▶ Using `as` to call that column `count_offense`
- ▶ Order from highest to lowest count of rows; take top 5

Step 2: adding row filtering to offenses in those top 5

```
1 select *
2 from caseinit
3 inner join (
4     select UPDATED_OFFENSE_CATEGORY as tmp_oc ,
5     count(*) as count_offense
6     from caseinit
7     where RACE in ("Black", "White")
8     group by UPDATED_OFFENSE_CATEGORY
9     order by count_offense desc
10    limit 5
11 ) as top5
12 on caseinit.UPDATED_OFFENSE_CATEGORY = top5.tmp_oc
```

Breaking it down:

- ▶ Put the query we wrote in previous step into a subquery
- ▶ The inner join means that the only rows from the caseinit table retained are ones where the UPDATED_OFFENSE_CATEGORY is in that top 5

Step 3: for each offense, get proportion diverted by race

```
1 select UPDATED_OFFENSE_CATEGORY, is_in_diversion, RACE,  
2 count(*) as count_divert, count(*)/count_group as prop_divert  
3 from caseinit  
4 inner join (  
5     select UPDATED_OFFENSE_CATEGORY as tmp_oc, RACE as tmp_race,  
6     count(*) as count_group  
7     from caseinit  
8     where RACE in ("Black", "White")  
9     group by UPDATED_OFFENSE_CATEGORY, RACE  
10    ) as tmp on tmp.tmp_race = caseinit.RACE  
11    and tmp.tmp_oc = caseinit.UPDATED_OFFENSE_CATEGORY  
12 group by UPDATED_OFFENSE_CATEGORY, RACE,  
13 is_in_diversion
```

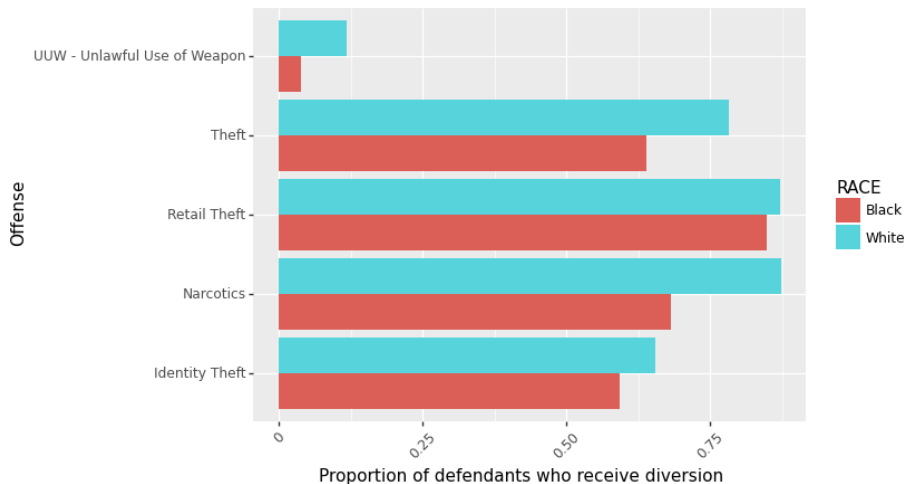
Logic:

- ▶ Filtering to Black and White defendants, **group by race and crime** to get the count of defendants in each race x crime combination (count_group)
- ▶ Merge retaining only defendants of those two races
- ▶ Group by **group by race, crime, and diversion status** to get count/proportion

Putting it together

```
1 select UPDATED_OFFENSE_CATEGORY, is_in_diversion, RACE,
2 count(*) as count_divert, count(*)/count_group as prop_divert
3 from caseinit
4 inner join (
5     select UPDATED_OFFENSE_CATEGORY as tmp_oc, RACE as tmp_race,
6     count(*) as count_group
7     from caseinit
8     where RACE in ("Black", "White")
9     group by UPDATED_OFFENSE_CATEGORY, RACE
10    ) as tmp on tmp.tmp_race = caseinit.RACE
11    and tmp.tmp_oc = caseinit.UPDATED_OFFENSE_CATEGORY
12 inner join (
13     select UPDATED_OFFENSE_CATEGORY as tmp_oc_t5, count(*) as
14     count_offense
15     from caseinit
16     where RACE in ("Black", "White")
17     group by UPDATED_OFFENSE_CATEGORY
18     order by count_offense desc
19     limit 5
20    ) as top5 on caseinit.UPDATED_OFFENSE_CATEGORY = top5.tmp_oc_t5
21 where is_in_diversion = 'True'
22 group by UPDATED_OFFENSE_CATEGORY, RACE,
23 is_in_diversion
```

After all that code, some disparities in narcotics



Activity 1: var creation and subquery practice

File: https://github.com/rebeccajohnson88/qss20_slides_activities/blob/main/activities/08_SQL_activity.ipynb

- ▶ Create a new column – `in_chicago`– that takes on the value of “YES” if `INCIDENT_CITY = Chicago`; “NO” otherwise (which represents incidents in Cook County suburbs outside the city limits)
- ▶ Use that column, along with the `is_in_diversion` column, to find the rate of diversions by whether the incident took place in Chicago or the suburbs
- ▶ Similarly, find the rate of diversions by location (city versus suburb indicator) and `RACE`

Outline

- ▶ SQL: ways of interacting with a database and starting connection
- ▶ Basics of rows and columns: selecting columns, selecting rows using logical conditions, and creating new columns based on conditions
- ▶ Subqueries, aggregations, and joins: one table
- ▶ **Subqueries, aggregations and joins: two tables**

Overview of diversion programs

DC: Drug Treatment Court. Twenty-four months of treatment-based probation focusing on connecting defendants with housing and employment opportunities. (*Post-Plea*)

DDPP: Drug Deferred Prosecution Program. Links low-level, non-violent drug offenders to community-based services and includes a formal substance abuse assessment. (*Pre-Plea*)

DS: Drug School. Four 2-and-a-half-hour lessons provided by licensed treatment providers with a focus on substance abuse and education, not treatment. (Ended in 2017) (*Post-Plea*)

RJCC: Restorative Justice Community Court. Community court located in North Lawndale that practices restorative justice, a system of criminal justice which focuses on the rehabilitation of offenders through reconciliation with victims and the community at large. For a case to be eligible for RJCC, the victim of the crime must agree to participate in the process. (*Pre-Plea*)

MHC: Mental Health Treatment Court. Twenty-four months of intensive probation focusing on treatment, housing, psychiatric stability, and employment services. (*Post-Plea*)

VC: Veterans Treatment Court. Twenty-four months of probation focusing on employment, housing, and any necessary treatment. (*Post Plea*)

Source: Cook County SAO data documentation

Merge with the case initiations data to explore things like

- ▶ How use of diversions differs across police departments (e.g., Chicago PD versus suburban PD)
- ▶ How bond/probation is related to diversion
- ▶ Age patterns (demographic var. available in caseinit but not in divert)

Left join of some cols from caseinit onto diversions: no aliasing

```
1 select divert.*,  
2 AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,  
3 INCIDENT_CITY  
4 from divert  
5 LEFT JOIN caseinit  
6 ON divert.CASE_ID = caseinit.CASE_ID  
7 AND divert.CASE_PARTICIPANT_ID = caseinit.  
   CASE_PARTICIPANT_ID
```

Breaking it down:

- ▶ Selected all cols from divert using the syntax tablename.*
- ▶ Selected only age, law enforc. agency, and incident city from caseinit

What happens if we select cols available in both dataframes?

```
1 select divert.*,  
2 AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,  
3 INCIDENT_CITY, RACE  
4 from divert  
5 LEFT JOIN caseinit  
6 ON divert.CASE_ID = caseinit.CASE_ID  
7 AND divert.CASE_PARTICIPANT_ID = caseinit.  
CASE_PARTICIPANT_ID
```

Error:

IntegrityError: 1052 (23000): Column 'RACE' in field list is ambiguous

How to fix: aliasing the col

```
1 select divert.*,  
2 AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,  
3 INCIDENT_CITY, caseinit.RACE as caseinit_race  
4 from divert  
5 LEFT JOIN caseinit  
6 ON divert.CASE_ID = caseinit.CASE_ID  
7 AND divert.CASE_PARTICIPANT_ID = caseinit.  
CASE_PARTICIPANT_ID
```

Breaking it down:

- Use syntax `tablename.colname as something` to alias the RACE var from the case initiations table as something else so that we know which table it's from

Simplifying the query by aliasing the table names

```
1 select d.*,  
2 AGE_AT_INCIDENT, LAW_ENFORCEMENT_AGENCY,  
3 INCIDENT_CITY, d.RACE as caseinit_race  
4 from divert as d  
5 LEFT JOIN caseinit as c  
6 ON d.CASE_ID = d.CASE_ID  
7 AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID
```

Breaking it down:

- ▶ Rename caseinit as c
- ▶ Rename diversions as d

Other joins

- ▶ INNER, OUTER, CROSS (latter takes all rows from LHS data and repeats each for all rows of RHS data, and vice versa)
- ▶ Good discussion here: <https://www.guru99.com/joins.html>

Combining aggregation of one table and join

Goal: among the cases that are diverted, for each of the charges (UPDATED_OFFENSE_CATEGORY) in the case initiations, find the percentage of defendants with that charge going to each DIVERSION_PROGRAM

Step 1: find the count of offenses by diversion program

```
1 select count(*) as count_offenses_byprogram ,  
2 UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM  
3 from divert as d  
4 INNER JOIN caseinit as c  
5 ON d.CASE_ID = c.CASE_ID  
6 AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID  
7 group by UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM  
8 order by count_offenses_byprogram desc
```


Step 1: find the count of offenses by diversion program

```
1 select count(*) as count_offenses_byprogram ,  
2 UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM  
3 from divert as d  
4 INNER JOIN caseinit as c  
5 ON d.CASE_ID = c.CASE_ID  
6 AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID  
7 group by UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM  
8 order by count_offenses_byprogram desc
```

Breaking it down:

- ▶ Joining divert to caseinit
- ▶ Grouping by both offense and diversion program
- ▶ Aggregating using count(*)

Step 2: find the count of offenses in general

```
1 select count(*) as count_offenses_total ,  
2 UPDATED_OFFENSE_CATEGORY  
3 from divert as d  
4 INNER JOIN caseinit as c  
5 ON d.CASE_ID = c.CASE_ID  
6 AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID  
7 group by UPDATED_OFFENSE_CATEGORY  
8 order by count_offenses_total desc
```

Step 3: combine into one query

```
1 select count(*) as count_offenses ,
2 count_offenses_byprogram/count(*) as
   prop_offenses_inprogram ,
3 UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
4 from caseinit
5 inner join (
6     select
7     count(*) as count_offenses_byprogram ,
8     UPDATED_OFFENSE_CATEGORY as ofc , DIVERSION_PROGRAM
9     from divert as d
10    INNER JOIN caseinit as c
11    ON d.CASE_ID = c.CASE_ID
12    AND d.CASE_PARTICIPANT_ID = c.CASE_PARTICIPANT_ID
13    group by UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM)
14    as num
15 on num.ofc = caseinit.UPDATED_OFFENSE_CATEGORY
16 group by UPDATED_OFFENSE_CATEGORY
17 order by prop_offenses_inprogram desc
```

Activity 2: join and subquery practice

- ▶ Use the following crosswalk and the CASE command to create a new variable `DIVERSION_PROGRAM_TEXT` that spells out the diversion programs
 - ▶ DC: Drug Court
 - ▶ DPPP: Drug Deferred Prosecution
 - ▶ DS: Drug School
 - ▶ RJCC: Restorative Justice
 - ▶ MHC: Mental Health Court
 - ▶ VC: Veteran Court
- ▶ Filter to Narcotics as the `UPDATED_OFFENSE_CATEGORY` and Black or White defendants
- ▶ Among defendants offered diversion (so defendants from `caseinit` also in the `diversions` table), the percent sent to each diversion program (separated by `RACE`)