PPOL564: Data Science I

Unit 09: Intro to SQL

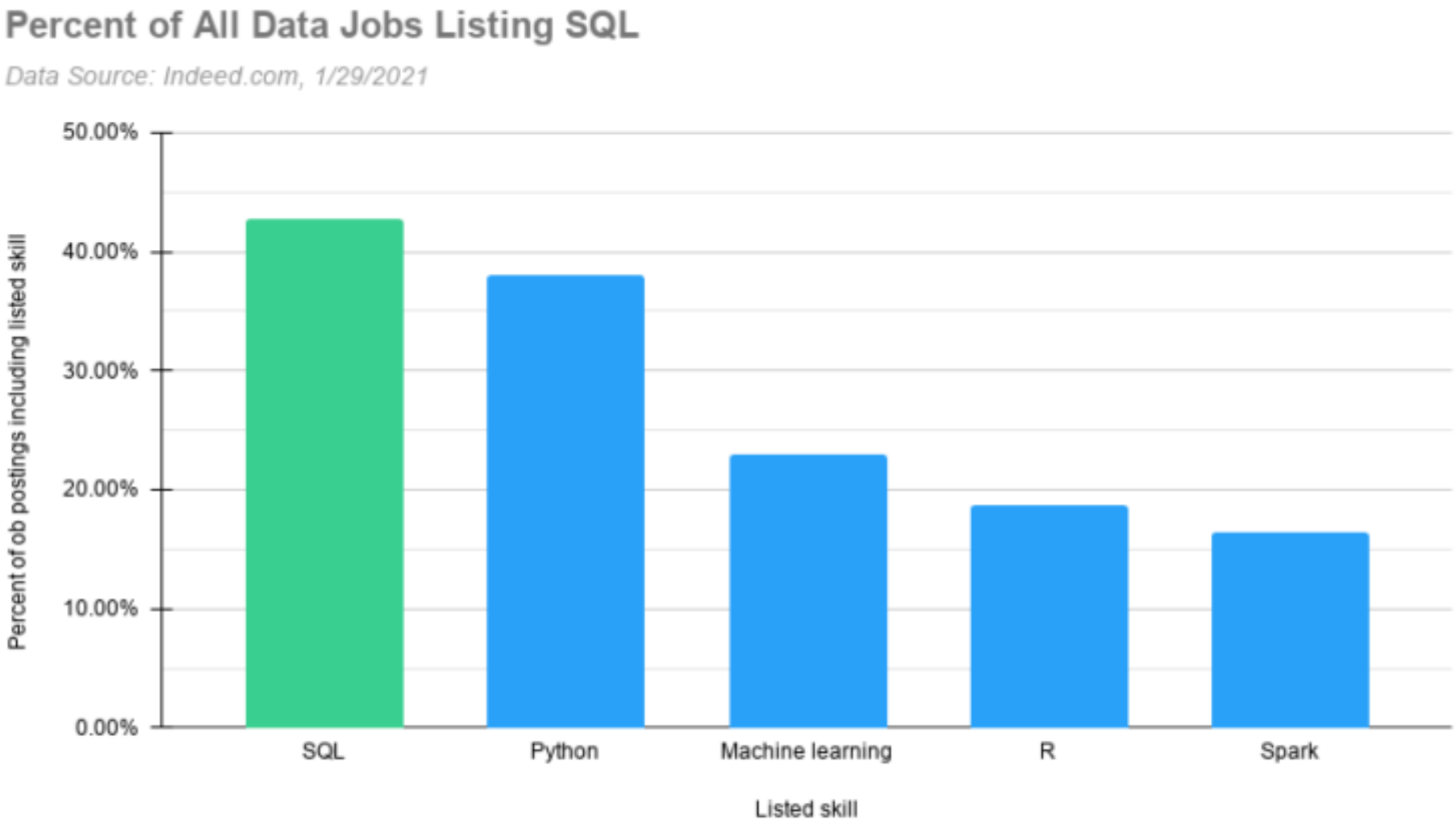
Outline

- **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?

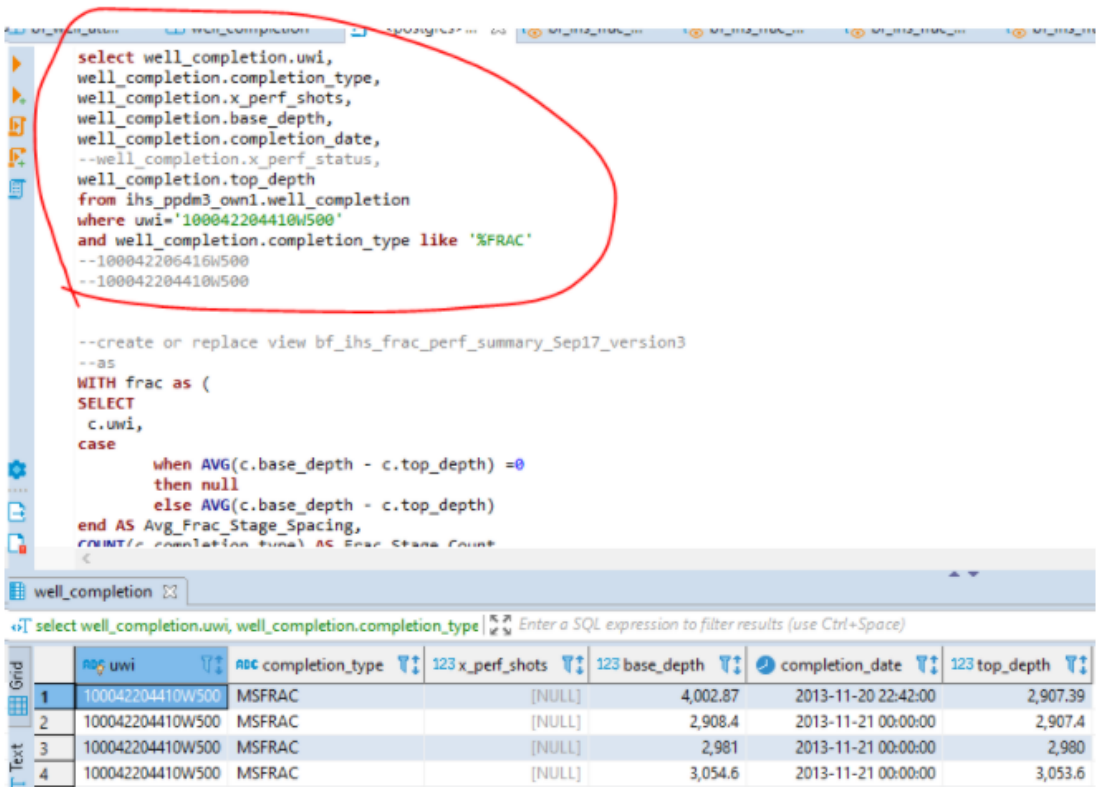
- **S**tructured**Q**uery**L**anguage
- While relatively uncommon in academia, many companies / governments expect data scientists to be able to write SQL queries
- In turn, a particular data warehouse/database might use different varieties of database engines to store data: Amazon Redshift; MySQL; postgreSQL; Microsoft SQL server; SQLite
- Nearly identical syntax but some small differences on the margins; here, we're using postgreSQL but similar syntax across engines

What is SQL and why might it be useful?



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:



```
select well_completion.uwi,
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_ppdm3_own1.well_completion
where uwi='100042204410W500'
and well_completion.completion_type like '%FRAC'
--100042206416W500
--100042204410W500


--create or replace view bf_ihs_frac_perf_summary_Sep17_version3
--as
WITH frac as (
SELECT
 c.uwi,
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,
```

well_completion

select well_completion.uwi, well_completion.completion_type | Enter a SQL expression to filter results (use Ctrl+Space)

	uwi	completion_type	x_perf_shots	base_depth	completion_date	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

Way of connecting here

- Database is setup in the `bit.io` cloud hosting service
- Python has a package `bitdotio` for establishing a connection to the database- see here for setup instructions: https://github.com/rebeccajohnson88/PPOL564_slides_activities/issues/65
- API key is here on pset6 assignment (using same database but different table for the pset): https://georgetown.instructure.com/courses/158038/assignments/814496
- Once you've established a connection, pandas has a command `pd.read_sql_query(insert query string)` that allows you to execute a SQL query and read in the result as a pandas dataframe

Preliminary step: load credentials and establish a connection

After (1) putting your database API key in your `cred.yml` file and (2) installing the bitdotio package, can run the following:

```
import bitdotio


creds = load_creds("../../../PPOL564_slides_activities/cred.yml")
b = bitdotio.bitdotio(creds['class_database']['api_key'])
cnx = b.get_connection("rebeccajohnson88/ppol564_classdb")
```

Working example: two tables from Chicago felony prosecution datasets used in pset 2

Desc.	Table	Main columns
Initiations	caseinit	CASE_ID; CASE_PARTICIPANT_ID; RACE; GENDER; UPDATED_OFFENSE_CATEGORY; is_in_diversion
Diversions	divert	CASE_ID; CASE_PARTICIPANT_ID; RACE; DIVERSION_PROGRAM; 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

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 5 rows from case initiations table

```
## define a query
sample_case_q = """
select *
from caseinit
limit 5
"""

## feed read sql query the query and my db connection
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 (`cnx`) to pandas `read_sql_query`

Columns: selecting specific columns with no transformations/additions

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

What this does: selects those two columns 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 operators:

- Equals: =
- Not equals: != (in other languages: <>)

Rows: filtering to specific rows using in or like

- Specify categories:

```
select CASE_ID, CASE_PARTICIPANT_ID,
RACE
from caseinit
where RACE in ('Black', 'HISPANIC')
```

- If contains Black anywhere in RACE string

```
select CASE_ID, CASE_PARTICIPANT_ID,
RACE
from caseinit
where RACE like '%Black%'
```

Columns: creating new columns based on conditions

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

```
select *,
CASE
    WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
    THEN 'Same offense'
    ELSE 'Diff offense'
END as charge_update
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):

```
select *,
CASE
    WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
    THEN 'Same offense'
    ELSE 'Diff offense'
END as charge_update
from caseinit
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 as charge_update
from caseinit
where charge_update = 'Diff offense'
': column "charge_update" does not exist
LINE 8: where charge_update = 'Diff offense'
```

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 two: 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

```
select *
from caseinit
inner join
    (select CASE_ID as cid,
    CASE_PARTICIPANT_ID as cpid,
    CASE
        WHEN OFFENSE_CATEGORY = UPDATED_OFFENSE_CATEGORY
        THEN 'Same offense'
        ELSE 'Diff offense'
    END as charge_update
    from caseinit) as tmp
    on tmp.cid = caseinit.case_ID and
    tmp.cpid = caseinit.CASE_PARTICIPANT_ID

where charge_update = "Diff offense"
```

Breaking things down, we use the parantheses 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 transformation or aggregation of the table
2. Join the result 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

```
select UPDATED_OFFENSE_CATEGORY,
count(*) as count_offense
from caseinit
where RACE in ('Black', 'White')
group by UPDATED_OFFENSE_CATEGORY
order by count_offense desc
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: nest that in a subquery to filter to rows where offense is in top 5

```sql
select *
from caseinit
inner join(
    select UPDATED_OFFENSE_CATEGORY as tmp_oc,
    count(*) as count_offense
    from caseinit
    where RACE in ('Black', 'White')
    group by UPDATED_OFFENSE_CATEGORY
    order by count_offense desc
    limit 5
    ) as top5
    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

Step 3: shift to writing code for the proportion sent to diversion outcome

```
select
avg(cast(is_in_diversion as INTEGER)) as prop_divert,
RACE, UPDATED_OFFENSE_CATEGORY
from caseinit
where race in ('Black', 'White')
group by race, UPDATED_OFFENSE_CATEGORY
order by prop_divert desc
```

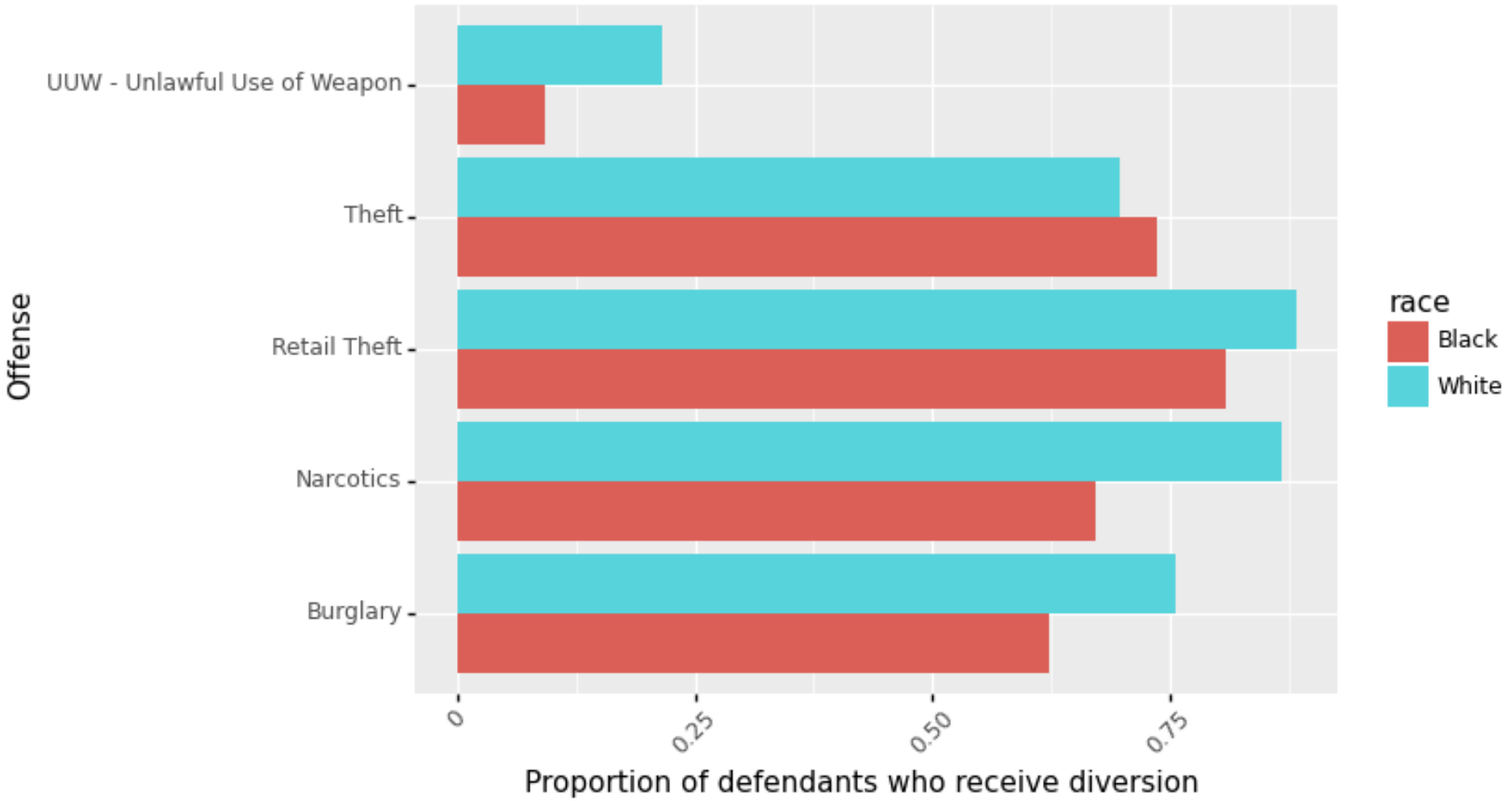
Logic:

- Filtering to Black and White defendants and grouping by race and offense type
- Recasting `is_in_diversion` as an integer - see here for details: https://www.postgresqltutorial.com/postgresql-tutorial/postgresql-cast/
- Similar to how, in pandas, we take the mean of a True/False column to get proportion true, we can take the mean of that binary indicator using the `avg` command to find the proportion diverted in each race/eth $\times$ type of offense group

Putting it together

```
select
avg(cast(is_in_diversion as INTEGER)) as prop_divert,
RACE, UPDATED_OFFENSE_CATEGORY
from caseinit
inner join (
    select UPDATED_OFFENSE_CATEGORY as tmp_oc_t5, count(*) as
    count_offense
    from caseinit
    where RACE in ('Black', 'White')
    group by UPDATED_OFFENSE_CATEGORY
    order by count_offense desc
    limit 5
    ) as top5 on caseinit.UPDATED_OFFENSE_CATEGORY = top5.tmp_oc_t5
where race in ('Black', 'White')
group by race, UPDATED_OFFENSE_CATEGORY
order by prop_divert desc
```

After all that code, some disparities



Activity break 1: var creation and subquery practice

Are elderly defendants more likely to receive diversion?

1. Create a new column – is_elderly when pulling from the caseinit table that takes on the value of 1 if the defendant's `AGE_AT_INCIDENT` is > 65; 0 otherwise
2. Use `where` to row filter to initiations where the defendant is elderly and use group by to find the count of cases diverted and not diverted (is_in_diversion); pull the table with those counts
3. Find the proportion of cases diverted for elderly versus non-elderly defendants (mean `is_in_diversion` by group

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

Can 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 (most 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
3 from divert
4 LEFT JOIN caseinit
5 ON divert.CASE_ID = caseinit.CASE_ID
6 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 RACE
3 from divert
4 LEFT JOIN caseinit
5 ON divert.CASE_ID = caseinit.CASE_ID
6 AND divert.CASE_PARTICIPANT_ID = caseinit.
    CASE_PARTICIPANT_ID
```

Error:

```
DatabaseError: Execution failed on sql '
select divert.*,
RACE
from divert
LEFT JOIN caseinit
ON divert.CASE_ID = caseinit.CASE_ID
AND divert.CASE_PARTICIPANT_ID = caseinit.CASE_PARTICIPANT_ID
': column reference "race" is ambiguous
LINE 3: RACE
        ^
```

How to fix: aliasing the col

```
select divert.*,
AGE_AT_INCIDENT,
caseinit.RACE as caseinit_race
from divert
LEFT JOIN caseinit
ON divert.CASE_ID = caseinit.CASE_ID
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,
3 c.RACE as caseinit_race
4 from divert as d
5 LEFT JOIN caseinit as c
6 ON d.CASE_ID = c.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.postgresqltutorial.com/postgresql-tutorial/postgresql-joins/

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 (get the numerator): find the count of offenses by diversion program

```
1 select count(*) as count_offenses_byprogram,
2 UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
3 from divert
4 INNER JOIN caseinit
5 ON divert.CASE_ID = caseinit.CASE_ID
6 AND divert.CASE_PARTICIPANT_ID = caseinit.
      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 (inner join to only keep diverted cases)
- Grouping by both offense and diversion program
- Aggregating using `count(*)`
- Ordering from highest to lowest count

Step 2 (get the denominator): 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
```

Similar logic as previous query but we're finding the total count of offenses aggregated across all diversion programs (so the denominator for the program-specific counts)

Step 3: combine into one query

```
select
count(*) as count_offenses,
count_offenses_byprogram,
DIVERSION_PROGRAM,
caseinit.UPDATED_OFFENSE_CATEGORY
from caseinit
inner join (select count(*) as count_offenses_byprogram,
UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM
from divert
INNER JOIN caseinit
ON divert.CASE_ID = caseinit.CASE_ID
AND divert.CASE_PARTICIPANT_ID = caseinit.CASE_PARTICIPANT_ID
group by UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM) as num
on num.UPDATED_OFFENSE_CATEGORY = caseinit.UPDATED_OFFENSE_CATEGORY
inner join (select CASE_ID as cid,
CASE_PARTICIPANT_ID as cpid
from divert
) as ppl_divert
on caseinit.CASE_ID = ppl_divert.cid
and caseinit.CASE_PARTICIPANT_ID = ppl_divert.cpid
group by caseinit.UPDATED_OFFENSE_CATEGORY, DIVERSION_PROGRAM,
count_offenses_byprogram
```

Activity break 2: join and subquery practice

- Create a new column: `is_vetcourt` where `DIVERSION_PROGRAM` == 'VC'; execute this query to make sure the query for this step is correct
- In the case initiations table, filter to (1) defendants with UPDATED_OFFENSE_CATEGORY is 'Narcotics'; (2) race is Black or White; and (3) is diverted; execute this query to make sure the query for this step is correct
- Combine the queries from step 1 and 2 to find, among the defendants diverted to something for narcotics offenses, the percentage of Black and percentage of white defendants sent specifically to veteran's treatment court