QSS20: Modern Statistical Computing

Unit 07: SQL

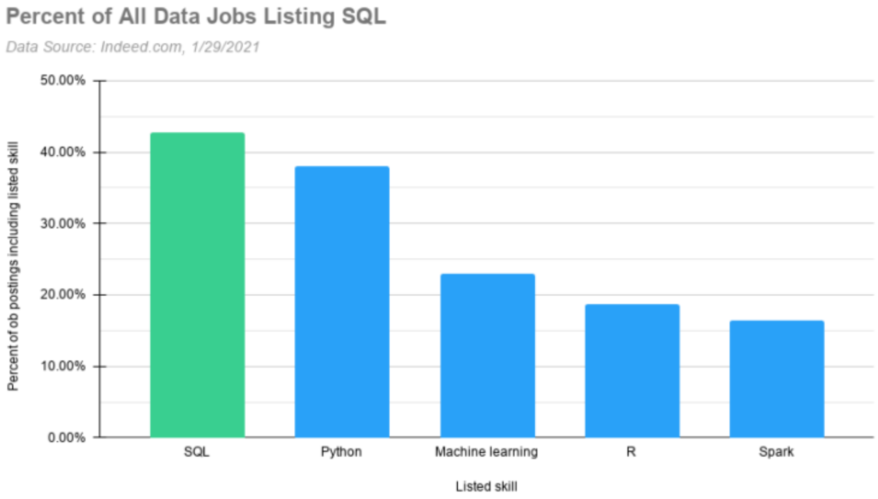
Outline

- **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?

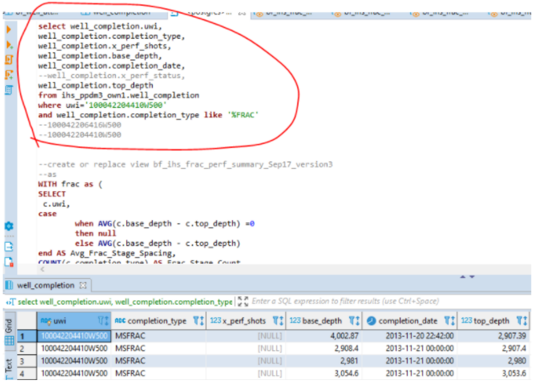
- **S**tructured**Q**uery**L**anguage
- While relatively uncommon in academia, many companies / governments expect data scientists to be able to write SQL queries
- In turn, a particular data warehouse/database might use different varieties of database engines to store data: Amazon Redshift; MySQL; postgreSQL;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

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:



```sql
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

Preliminary step: load credentials and establish a connection

```python
## import mysql connector
import mysql.connector

## load creds
creds = load_creds("../private_data/creds_forclass.yaml")

## use username, pwd, host, port, etc
## to establish a connection to the database
cnx = mysql.connector.connect(
     user=creds['practice_database']['db_user'],
     password=creds['practice_database']['db_password'],
     port=creds['practice_database']['port'],
     database= creds['practice_database']['database'],
     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; UPDATED_OFFENSE_CATEGORY; is_in_diversion	rjohnson
Diversions	divert	CASE_ID; CASE_PARTICIPANT_ID; RACE; DIVERSION_PROGRAM; OFFENSE_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
```

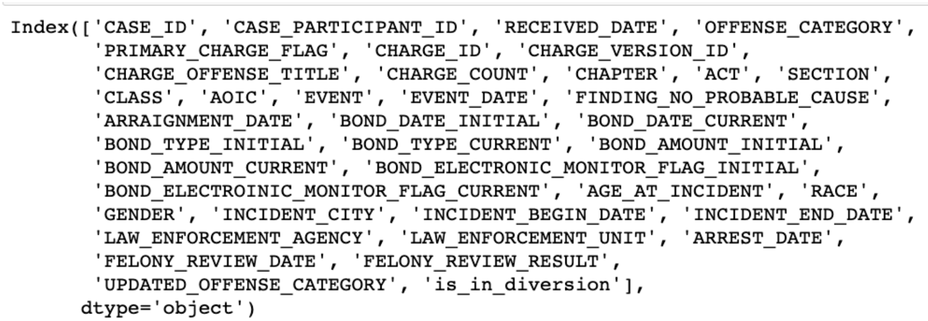
Examining structure of data: selecting first 10 rows from case initiations table

```python
## define a query
sample_case_q = """
select *
from caseinit
limit 10
"""
## 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 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

```sql
select CASE_ID, CASE_PARTICIPANT_ID
from caseinit
```

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

Rows: filtering to specific rows using where

```sql
select CASE_ID, CASE_PARTICIPANT_ID,
AGE_AT_INCIDENT
from caseinit
where AGE_AT_INCIDENT > 40
```

Other logical operations:

- Equals: =
- Not equals: <>

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 charge_update
from caseinit
```

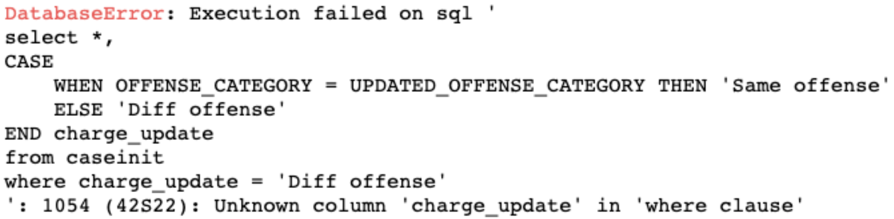
What if we want to create a new col and then filter using that same columns as part of the same query? Query

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 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 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

```
select *
from caseinit
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

```
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 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 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

```
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: adding row filtering to offenses in those top 5

```
select *
from caseinit
inner join(
    select UPDATED_OFFENSE_CATEGORY as tmp_oc,
    count(*) as count_offense
    from caseinit
    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 top 5

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

```sql
select UPDATED_OFFENSE_CATEGORY, is_in_diversion , RACE,
count(*) as count_divert , count(*)/count_group as prop_divert
from caseinit
inner join (
     select UPDATED_OFFENSE_CATEGORY as tmp_oc, RACE as tmp_race ,
     count(*) as count_group
     from caseinit
     where RACE in ("Black" , "White")
     group by UPDATED_OFFENSE_CATEGORY, RACE
     ) as tmp on tmp.tmp_race = caseinit.RACE
     and tmp.tmp_oc = caseinit.UPDATED_OFFENSE_CATEGORY
group by UPDATED_OFFENSE_CATEGORY, RACE,
is_in_diversion
```

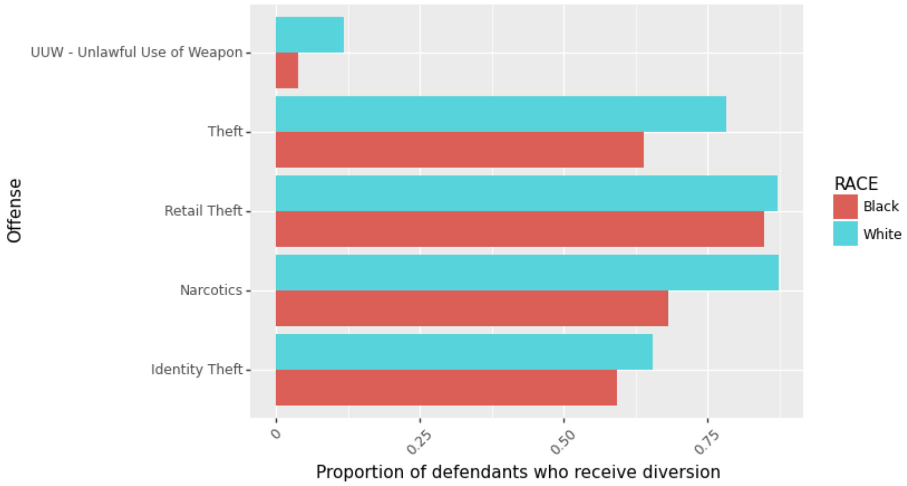
Logic:

- 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

```sql
select UPDATED_OFFENSE_CATEGORY, is_in_diversion, RACE,
count(*) as count_divert, count(*)/count_group as prop_divert
from caseinit
inner join (
    select UPDATED_OFFENSE_CATEGORY as tmp_oc, RACE as tmp_race,
    count(*) as count_group
    from caseinit
    where RACE in ("Black", "White")
    group by UPDATED_OFFENSE_CATEGORY, RACE
    ) as tmp on tmp.tmp_race = caseinit.RACE
    and tmp.tmp_oc = caseinit.UPDATED_OFFENSE_CATEGORY
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 is_in_diversion = 'True'
group by UPDATED_OFFENSE_CATEGORY, RACE,
is_in_diversion
```

After all that code, some disparities in narcotics



Short practice

- 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`