SQL

Lukas Hager

2024-05-06

Learning Objectives

- Be able to use python to connect to a database
- Understand the basics of SQL syntax and how to query data

SQL Background

Pronunciation[1]

"Sequel", not "Ess-Queue-Ell"

[1] Some of this material is based off of material created by Dr. Konstantin Golyaev.

Background

- Structured Query Language
- Used to extract data from relational databases
- Core concepts:
  - Record (table row)
  - Table column
  - Table – collection of rows and columns

Tables

- Any table operation produces another table as a result
- A record is a collection of key-value pairs
  - Think of this like a row in an excel spreadsheet
  - "Name": "John", "Salary": 50, ...
- A table is a collection of records
  - "Name": "John", "Salary": 50, ...,
  - "Name": "Mary", "Salary": 55, ...
- pandas treats tables as `DataFrames`

SQL in Pipelines

- Most large companies and research groups store data in relational databases
- The first step of any project is to define the data you need and query it from SQL
- Once you have the data, you can clean and model using `pandas` etc.
- This first querying step is key!

Why?

Why should you use databases instead of CSVs?

Answer (per Luke[2] Wylie[3])

1. Databases are tools built specifically for using and sharing data in a matched "state" - as soon as someone else needs to use your data at the same time as you, and even keeping track of changes and mutations to the transaction, a CSV is useless.
2. As soon as you start mutating data and creating multiple datasets while refusing to use a database, you resign yourself to the special hell that is juggling multiple CSVs. You will inevitably lose data.

[2] Senior Data Engineer/Data God at Microsoft
[3] My neighbor

Accessing a Database in Python

`sqlalchemy`

- There are lots of ways to connect to a database
  - Hopefully the group that you're working with already has an in-house solution
- We'll work with a very simple version (no authentication, etc.)
- Connecting to a `sqlite` database using `sqlalchemy`

create_engine

- We have a .db file called auctions.db that contains data on bidding for 500 North Face clothing items on ShopGoodwill.com[4]
- To connect to it, we have to create a sqlalchemy engine:

```
import sqlalchemy
from sqlalchemy import create_engine

path = '/Users/hlukas/git/personal_website/static/econ-481/dat
engine = create_engine(f'sqlite:///{path}')
```

[4] I'm using data scraped from this site in my general exam paper, so let me know if you see anything interesting in it!

`create_engine` Argument

- Note that at the beginning, we tell `sqlalchemy` what sort of database we're connecting to
- We then pass three / characters before the database location.

Listing Tables

- Databases contain multiple tables
- We want to know what they are

```
from sqlalchemy import inspect

inspector = inspect(engine)
inspector.get_table_names()
```

```
['bids', 'items']
```

Listing Tables

- Databases contain multiple tables
- We want to know what they are

```python
from sqlalchemy import inspect

inspector = inspect(engine)
inspector.get_table_names()
```

```
['bids', 'items']
```

So we have two tables, named “bids” and “items”

Querying Data

- We'll begin by working with SQL in a "traditional" sense, where we just write queries instead of leveraging the python package
  - Libraries like `sqlalchemy` or `pyspark` have methods to take the place of querying
  - These are a little easier to learn once we get the basics of writing a query
- Query: a letter to the database telling it what we want

Writing a Query Class

To assess the output of our queries, we're going to write a class that will run our query against the database and return a DataFrame as the table output.

```
import pandas as pd
from sqlalchemy.orm import Session

class DataBase:
    def __init__(self, loc: str, db_type: str = "sqlite") -> N
        """Initialize the class and connect to the database""
        self.loc = loc
        self.db_type = db_type
        self.engine = create_engine(f'{self.db_type}:///{self
    def query(self, q: str) -> pd.DataFrame:
        """Run a query against the database and return a DataF
        with Session(self.engine) as session:
            df = pd.read_sql(q, session.bind)
        return(df)

auctions = DataBase(path)
```

Aside: Why a Class?

- Why is a class better than a function here?

Aside: Why a Class?

- Why is a class better than a function here?

- A function would either require us to pass the engine as an argument or reference a global variable (not good)
- In the class, all of our queries will share the same engine
- Logical flow – we create run queries against only one database at a time

Queries

Query Syntax

- SELECT *comma-separated list of columns*
- FROM *Table1 JOIN Table2 ... JOIN TableN*
- WHERE *Condition1 AND ... AND ConditionM*
- GROUP BY *comma-separated list of grouping columns*
- [HAVING] *Condition1 AND ... AND ConditionK*
- [ORDER BY] *comma-separated list of sorting cols*
- [LIMIT] *number of rows to return*

SELECT * Statement[5]

```python
q = 'select * from bids'
print(auctions.query(q).head())
```

```
   index  bidLogId     itemId  itemPrice  bidAmount  \
0     50         0  178348858       9.99       20.0
1     51         0  178348858      13.00       12.0
2     52         0  178348858      21.00       23.0
3     53         0  178348858      24.00       35.0
4     54         0  178348858      36.00       48.0

                     bidTime  quantity bidIPAddress adCode serv
0  2023-09-18 16:11:04.587000         1         None   None
1  2023-09-22 14:22:06.700000         1         None   None
2  2023-09-23 12:35:18.157000         1         None   None
3  2023-09-23 18:23:27.993000         1         None   None
4  2023-09-23 18:37:47.213000         1         None   None

   retracted bidderName highBidderName  isBuyerHighBidder  isLog
0          0     a****9         a****9                  0
1          0     S****p         a****9                  0
```

SELECT Columns Statement

```
q = 'select itemid, description, isbuynowused from items'
print(auctions.query(q).head())
```

```
      itemId                                        description
0  179353985  <p><strong>Description:</strong></p>\n<p>Women...
1  177087535  <p><strong>Details &amp; Condition</strong></p...
2  180876361  <p>The North Face Womens Pink Long Sleeve Mock...
3  177763109  <p><br></p><ul><li><span class="ql-size-large"...
4  179660197  <p><b>Title: </b>The North Face Mens Red Flat ...
```

JOIN Statements

Recall our discussion on joining in pandas – these are SQL-style joins, and SQL has the same types.

```
q = """
select items.itemid, items.description, bids.biddername, bids
from items
left join bids
on items.itemid = bids.itemid
"""
print(auctions.query(q).head())
```

```
      itemId                                        description
0  179353985  <p><strong>Description:</strong></p>\n<p>Women...
1  177087535  <p><strong>Details &amp; Condition</strong></p...
2  180876361  <p>The North Face Womens Pink Long Sleeve Mock...
3  177763109  <p><br></p><ul><li><span class="ql-size-large"...
4  177763109  <p><br></p><ul><li><span class="ql-size-large"...

   bidAmount                     bidTime
0        NaN                        None
```

JOIN Aliases

Should this run?

```python
q = """
select itemid, description, biddername, bidamount, bidtime
from items
left join bids
on items.itemid = bids.itemid
"""
print(auctions.query(q).head())
```

```
OperationalError: (sqlite3.OperationalError) ambiguous column na
[SQL:
select itemid, description, biddername, bidamount, bidtime
from items
left join bids
on items.itemid = bids.itemid
]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

JOIN Renaming Tables

It's often convenient to rename tables in joins to make your query less verbose (potentially at the cost of readability)

```
q = """
select i.itemid, i.description, b.biddername, b.bidamount, b.l
from items as i
left join bids as b
on i.itemid = b.itemid
"""
print(auctions.query(q).head())
```

```
      itemId                                          description
0  179353985  <p><strong>Description:</strong></p>\n<p>Women...
1  177087535  <p><strong>Details &amp; Condition</strong></p...
2  180876361  <p>The North Face Womens Pink Long Sleeve Mock...
3  177763109  <p><br></p><ul><li><span class="ql-size-large"...
4  177763109  <p><br></p><ul><li><span class="ql-size-large"...

   bidAmount                   bidTime
0        NaN                      None
```

Exercise: Joins

For each of the join types supported in `sqlite` (left, inner, cross), perform the join on the two tables and report the number of observations in the resulting join.

Solutions: Joins

```
join_types = ['inner', 'left', 'cross']
queries = [
    f"""select count(*) as n
    from items as i
    {join} join bids as b
    on i.itemid = b.itemid""" for join in join_types
]
[auctions.query(q)['n'].item() for q in queries]
```

```
[551, 879, 551]
```

WHERE

```python
q = """
select i.itemid, i.description, b.biddername, b.bidamount, b.b
from items as i
left join bids as b
on i.itemid = b.itemid
where b.bidamount is not null
"""
print(auctions.query(q).head())
```

```
      itemId                                        description
0  178348858  <p><br></p><ul><li><span class="ql-size-large"...
1  178348858  <p><br></p><ul><li><span class="ql-size-large"...
2  178348858  <p><br></p><ul><li><span class="ql-size-large"...
3  178348858  <p><br></p><ul><li><span class="ql-size-large"...
4  178348858  <p><br></p><ul><li><span class="ql-size-large"...

   bidAmount                     bidTime
0       20.0  2023-09-18 16:11:04.587000
1       12.0  2023-09-22 14:22:06.700000
2       23.0  2023-09-23 12:35:18.157000
```

WHERE With Multiple Conditions

```
q = """
select i.itemid, i.description, b.biddername, b.bidamount, b.b
from items as i
left join bids as b
on i.itemid = b.itemid
where b.bidamount is not null and i.isbuynowused is false
"""
print(auctions.query(q).head())
```

```
      itemId                                        description
0  180876361  <p>The North Face Womens Pink Long Sleeve Mock...
1  177763109  <p><br></p><ul><li><span class="ql-size-large"...
2  177763109  <p><br></p><ul><li><span class="ql-size-large"...
3  177763109  <p><br></p><ul><li><span class="ql-size-large"...
4  177763109  <p><br></p><ul><li><span class="ql-size-large"...

   bidAmount                     bidTime
0      19.99  2023-10-18 05:54:55.327000
1      10.00  2023-09-17 11:52:27.447000
2      14.00  2023-09-17 17:33:48.517000
```

GROUP BY

The same as `.groupby()` in pandas – add aggregating functions to the `SELECT` clause

```
q = """
select i.itemid, count(distinct b.biddername) as n_bidders
from items as i
left join bids as b
on i.itemid = b.itemid
where b.bidamount is not null and i.isbuynowused is false
group by i.itemid
"""
print(auctions.query(q).head())
```

```
      itemId  n_bidders
0  165561698          1
1  170983900          1
2  172998011          2
3  173907435          1
4  174445924          3
```

Aside: COUNT

We can also just count observations without a grouping:

```
q = """
select count(*) from items
"""
print(auctions.query(q).head())
```

```
   count(*)
0       500
```

Aside: COUNT

We can also just count observations without a grouping:

```
q = """
select count(*) from items
"""
print(auctions.query(q).head())
```

```
   count(*)
0       500
```

Or count the distinct number of something without a grouping:

```
q = """
select count(distinct biddername) from bids
"""
print(auctions.query(q).head())
```

```
   count(distinct biddername)
0                         284
```

Exercise: `MIN` and `MAX`

In SQL, `MIN` and `MAX` are aggregating functions that work the same way as `COUNT`. Use them to create a table of the number of bids each bidder submitted for each item, as well as their largest and smallest bid.

Exercise: MIN and MAX

```
q = """
select itemid, biddername, count(*) as n_bids, min(bidamount) as min_bid,
max(bidamount) as max_bid
from bids
group by itemid, biddername
"""
print(auctions.query(q).head())
```

```
      itemId bidderName  n_bids  min_bid  max_bid
0  165561698     n****4       1     9.91     9.91
1  170983900     c****3       1     9.91     9.91
2  172998011     A****e       1     9.91     9.91
3  172998011     J****m       1     9.91     9.91
4  173907435     M****n       1    14.99    14.99
```

Filter on Aggregate Function Value

What if we only care about bid distribution for a bidder when their largest bid is more than $20?

```
q = """
select itemid, biddername, count(*) as n_bids, min(bidamount)
max(bidamount) as max_bid
from bids
group by itemid, biddername
where max_bid > 20
"""
print(auctions.query(q).head())
```

```
OperationalError: (sqlite3.OperationalError) near "where": synta
[SQL:
select itemid, biddername, count(*) as n_bids, min(bidamount) as
max(bidamount) as max_bid
from bids
group by itemid, biddername
where max_bid > 20
]
```

HAVING

If we want to filter on the aggregate function value, we need to use `HAVING` instead of `WHERE`

```
q = """
select itemid, biddername, count(*) as n_bids, min(bidamount)
max(bidamount) as max_bid
from bids
group by itemid, biddername
having max_bid > 20
"""
print(auctions.query(q).head())
```

```
      itemId bidderName  n_bids  min_bid  max_bid
0  174767945     C****2       3    24.44    34.00
1  174767945     b****z       4    25.00    33.00
2  174871788     J****3       1    21.00    21.00
3  174871788     v****l       3    15.00    22.00
4  174901466     c****8       1    39.99    39.99
```

ORDER BY

Sorting works in an intuitive way

```
q = """
select itemid, biddername, count(*) as n_bids, min(bidamount)
max(bidamount) as max_bid
from bids
group by itemid, biddername
having max_bid > 20
order by max_bid desc, biddername
"""
print(auctions.query(q).head())
```

```
      itemId bidderName  n_bids  min_bid  max_bid
0  180573534     j****a       1    301.0    301.0
1  180573534     A****3       4    140.0    300.0
2  180601736     c****c       4    180.0    201.0
3  180601736     A****8       2    150.0    200.0
4  180601736     B****a       1    160.0    160.0
```

LIMIT

We've been asking for the head of our DataFrame to limit output – we can do this directly in the query:

```
q = """
select itemid, biddername, count(*) as n_bids, min(bidamount)
max(bidamount) as max_bid
from bids
group by itemid, biddername
having max_bid > 20
order by max_bid desc, biddername
limit 1
"""
print(auctions.query(q))
```

```
      itemId bidderName  n_bids  min_bid  max_bid
0  180573534     j****a       1    301.0    301.0
```

Exercise: Bidder Participation

In our sample, how many bidders participate in multiple auctions? And how many auctions do they participate in?

Solutions: Bidder Participation

```
q = """
select biddername, count(distinct itemid) as n_auctions
from bids
group by biddername
having n_auctions > 1
"""
bidder_participation = auctions.query(q)
print(bidder_participation.shape[0])
```

60

Solutions: Bidder Participation

We'll see soon that we could also do this with a "subquery"

```
q = """
select count(*) from (
    select biddername, count(distinct itemid) as n_auctions
    from bids
    group by biddername
    having n_auctions > 1
) as a
"""
print(auctions.query(q))
```

```
   count(*)
0        60
```

Solutions: Bidder Participation

```
import numpy as np
bidder_participation.hist(
    bins = np.arange(
        np.min(bidder_participation['n_auctions']),
        np.max(bidder_participation['n_auctions'])+1
    )
);
```

Window Functions

OVER

If we want to compute operations by group and assign it as a new variable, we need to tell SQL how to organize the groups:

```
q = """
select itemid, min(bidamount) over (partition by itemid) as mi
from bids
"""
print(auctions.query(q).head())
```

```
      itemId  min_bid  itemPrice
0  165561698     9.91       9.91
1  170983900     9.91       9.91
2  172998011     9.91       9.91
3  172998011     9.91       9.91
4  173907435    14.99      14.99
```

LAG

Window functions are particularly useful if we need to lag data in SQL

```
q = """
select itemid,
min(bidamount) over (partition by itemid) as min_bid,
itemprice,
lag(itemprice) over (partition by itemid order by bidtime) as
from bids
"""
print(auctions.query(q).head())
```

```
      itemId  min_bid  itemPrice  lagged_price
0  165561698     9.91       9.91           NaN
1  170983900     9.91       9.91           NaN
2  172998011     9.91       9.91           NaN
3  172998011     9.91       9.91          9.91
4  173907435    14.99      14.99           NaN
```

Creating Columns

String Concatenation

String concatenation in SQL is performed with ||

```
q = """
select title, itemid, title || " " || description as full_des
from items
"""
print(auctions.query(q).head())
```

```
                                         title     itemId
0        Womens Size M The North Face Zip Up Jacket  179353985
1  The North Face Women's Size 4 Tan/Khaki Lightw...  177087535
2  The North Face Womens Pink Long Sleeve Mock Ne...  180876361
3  The North Face Women's Medium Sweaters/Shirt L...  177763109
4  The North Face Mens Red Flat Front Slash Pocke...  179660197

                                    full_description
0  Womens Size M The North Face Zip Up Jacket <p>...
1  The North Face Women's Size 4 Tan/Khaki Lightw...
2  The North Face Womens Pink Long Sleeve Mock Ne...
3  The North Face Women's Medium Sweaters/Shirt L...
```

Arithmetic

```
q = """
select itemid, currentprice, shipping,
currentprice + shipping as final_price
from items
"""
print(auctions.query(q).head())
```

```
      itemId  currentPrice  shipping  final_price
0  179353985         10.99         0        10.99
1  177087535         24.98         0        24.98
2  180876361         19.99         0        19.99
3  177763109         15.00         0        15.00
4  179660197         12.99         0        12.99
```

CASE WHEN

SQL's if-else statement (similar to R's `ifelse` or `case_when` verbs)

```
q = """
select itemid, currentprice, shipping,
currentprice + case when shipping == 0 then 5 else shipping en
from items
order by shipping desc
"""
print(auctions.query(q).head())
```

```
      itemId  currentPrice  shipping  final_price
0  176705357         19.99         2        21.99
1  179025543         14.99         2        16.99
2  179353985         10.99         0        15.99
3  177087535         24.98         0        29.98
4  180876361         19.99         0        24.99
```

More Cases

We can use `LIKE` to pattern match – % means zero, one, or multiple characters (this is a bad application – why?)

```
q = """
select itemid, currentprice,
case when lower(description) like "%small%" then "small"
when lower(description) like "%medium%" then "medium"
when lower(description) like "%large%" then "large"
else null end as size
from items
where size is not null
"""
print(auctions.query(q).head())
```

```
      itemId  currentPrice   size
0  177087535         24.98  small
1  180876361         19.99  small
2  177763109         15.00  large
3  179660197         12.99  small
4  176601978          9.99  large
```

Database Operations

Adding to our Class

- SQL doesn't just query data – it also allows us to change the database
  - We can add tables (temporary or otherwise), for example
- We want to be able to also run statements that don't just return data, but perform operations on our database
- Let's add an `execute` method that facilitates this for our `engine`

New Class

```
from sqlalchemy import text

class DataBase:
    def __init__(self, loc: str, db_type: str = "sqlite") -> N
        """Initialize the class and connect to the database""
        self.loc = loc
        self.db_type = db_type
        self.engine = create_engine(f'{self.db_type}:///{self
    def query(self, q: str) -> pd.DataFrame:
        """Run a query against the database and return a DataF
        with Session(self.engine) as session:
            df = pd.read_sql(q, session.bind)
        return(df)
    def execute(self, q: str) -> None:
        """Execute statement on the database"""
        with self.engine.connect() as conn:
            conn.execute(text(q))

auctions = DataBase(path)
```

Creating a Joined Table

If we want to create a new table that contains only observations with bids where the buy now option wasn't used, we can execute a statement to do so.

```
q = """
create table full_data as
select i.*, b.*
from items as i
inner join bids as b
on i.itemid = b.itemid
where i.isbuynowused = 0
"""
auctions.execute("drop table if exists full_data")
auctions.execute(q)
print(auctions.query("select * from full_data limit 1"))
```

```
   index buyerCountry buyerCountryCode buyerState buyerStreet bu
0  12100         None               US       None        None

                                   categoryParentList defaultShipping
```

Dropping Tables

Why do we need the first statement? Because SQL won't let us create a table that already has a given name

```
q = """
create table full_data as
select * from items
"""
auctions.execute(q)
```

```
OperationalError: (sqlite3.OperationalError) table full_data alr
[SQL:
create table full_data as
select * from items
]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

Temporary Tables Creation

```
q = """
create temp table full_data as
select i.*, b.*
from items as i
inner join bids as b
on i.itemid = b.itemid
where i.isbuynowused = 0
"""
auctions.execute("drop table if exists full_data")
auctions.execute(q)
print(auctions.query("select * from full_data limit 1"))
```

```
   index buyerCountry buyerCountryCode buyerState buyerStreet bu
0  12100         None               US       None        None

                              categoryParentList defaultShipping
0  10|Clothing|27|Women's Clothing|154|Outerwear

                                          description  \
0  <p>The North Face Womens Pink Long Sleeve Mock
```

Rerunning

```
auctions = DataBase(path)
print(auctions.query("select * from full_data limit 1"))
```

```
OperationalError: (sqlite3.OperationalError) no such table: full
[SQL: select * from full_data limit 1]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

Rerunning

```
auctions = DataBase(path)
print(auctions.query("select * from full_data limit 1"))
```

```
OperationalError: (sqlite3.OperationalError) no such table: full
[SQL: select * from full_data limit 1]
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```

- Temporary tables get dropped when a session or connection is closed
- This is desirable if these are just intermediate tables (they won't clog up your database)
- This is undesirable if they take a lot of time to compute (maybe just save them as normal tables)

Exercise: Temporary Tables

For each bid, express its time as relative to when the auction ended (`endtime`). That means that if an auction was 10 hours long (as measured by `endtime - starttime`) and a bid was placed an hour before the auction ended, it would have a normalized timestamp of .1. Plot this distribution as a histogram.

Hint: to compute the difference in time between two dates, use `julianday(time1)-julianday(time2)`.

Solutions: Temporary Tables

```
q = """
create temp table auction_length as
select itemid, starttime, endtime,
julianday(endtime) - julianday(starttime) as length
from items
"""
auctions.execute("drop table if exists auction_length")
auctions.execute(q)
print(auctions.query('select * from auction_length limit 4'))
```

```
     itemId                   startTime                   end
0  179353985  2023-09-28 17:00:54.000000  2023-10-02 18:14:00.00
1  177087535  2023-09-04 22:54:00.000000  2023-09-12 19:46:00.00
2  180876361  2023-10-14 03:18:40.000000  2023-10-19 04:04:40.00
3  177763109  2023-09-12 08:22:45.000000  2023-09-17 18:34:00.00
```

Solutions: Temporary Tables

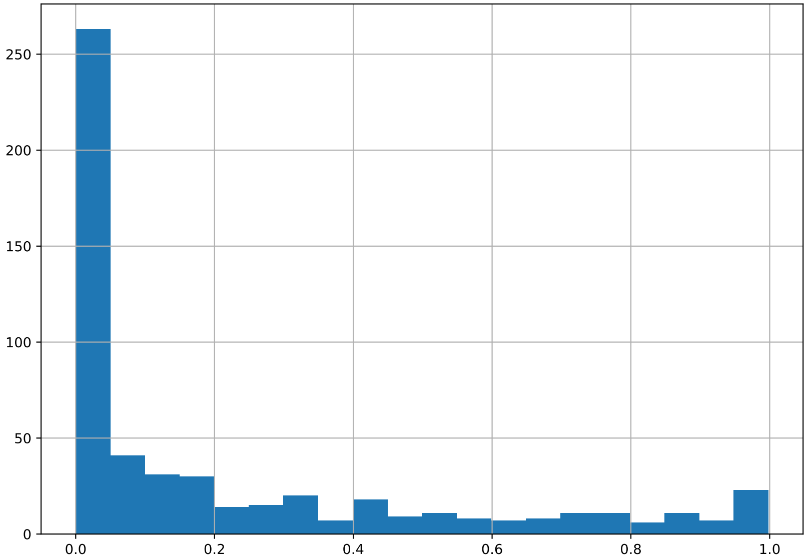
```
q = """
select b.itemid, b.bidtime, a.starttime, a.endtime,
(julianday(endtime)-julianday(bidtime)) / a.length as time_no
from bids as b
inner join auction_length as a
on b.itemid=a.itemid
"""
df = auctions.query(q)
print(df.head())
```

```
      itemId                     bidTime                      start
0  178348858  2023-09-18 16:11:04.587000  2023-09-18 14:29:56.00
1  178348858  2023-09-22 14:22:06.700000  2023-09-18 14:29:56.00
2  178348858  2023-09-23 12:35:18.157000  2023-09-18 14:29:56.00
3  178348858  2023-09-23 18:23:27.993000  2023-09-18 14:29:56.00
4  178348858  2023-09-23 18:37:47.213000  2023-09-18 14:29:56.00

                     endTime  time_norm
0  2023-09-23 18:39:00.000000   0.986422
1  2023-09-23 18:39:00.000000   0.227799
```

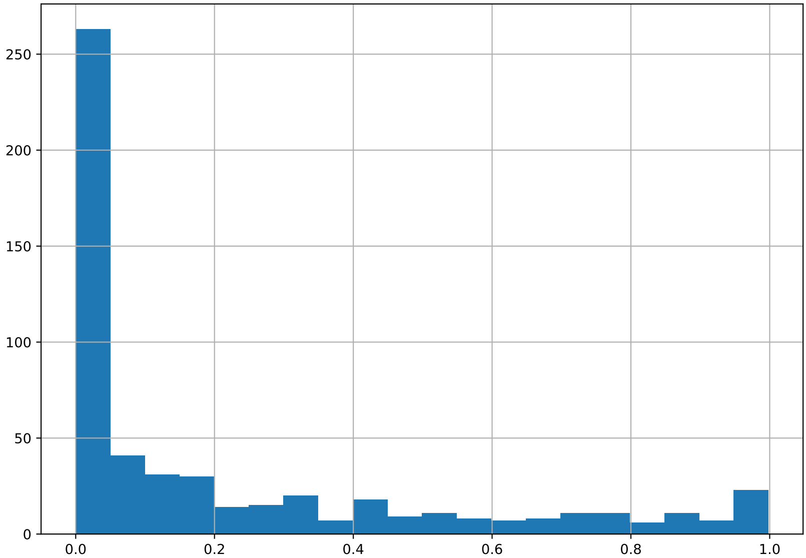
Solutions: Temporary Tables

```
df['time_norm'].hist(bins=20)
```



Solutions: Temporary Tables

```
df['time_norm'].hist(bins=20)
```



Subqueries

Alternative Solution

```
q = """
select b.itemid, b.bidtime, a.starttime, a.endtime,
(julianday(endtime)-julianday(bidtime)) / a.length as time_no
from bids as b
inner join (
    select itemid, starttime, endtime,
    julianday(endtime) - julianday(starttime) as length
    from items
) as a
on b.itemid=a.itemid
"""
df = auctions.query(q)
print(df.head(2))
```

```
      itemId                      bidTime                       start
0  178348858  2023-09-18 16:11:04.587000  2023-09-18 14:29:56.00
1  178348858  2023-09-22 14:22:06.700000  2023-09-18 14:29:56.00

                      endtime  time_norm
0  2023-09-23 18:39:00.000000   0.986422
```

Better Approach

Using WITH improves readability

```
q = """
with a as (
    select itemid, starttime, endtime,
    julianday(endtime) - julianday(starttime) as length
    from items
)
select b.itemid, b.bidtime, a.starttime, a.endtime,
(julianday(endtime)-julianday(bidtime)) / a.length as time_no
from bids as b
inner join a
on b.itemid=a.itemid
"""
df = auctions.query(q)
print(df.head(2))
```

```
      itemId                     bidTime                    start
0  178348858  2023-09-18 16:11:04.587000  2023-09-18 14:29:56.00
1  178348858  2023-09-22 14:22:06.700000  2023-09-18 14:29:56.00
```

Do As I Say, Not As I Do

Writing Readable SQL Queries[6]

No unified linting tools such as `pylint` for python

- SQL is NOT case sensitive and ignores whitespace
- It is easy to write unreadable code

Always assume that the code you write today will be inherited by a murderous psychopath who knows where you live!

[6] All of this advice comes directly from Dr. Konstantin Golyaev's slides.

Use Consistent Indentation/Breaks

```
SELECT <X>
FROM <A>
WHERE <TRUE>
```

not

```
SELECT <X> FROM <A> WHERE <TRUE>
```

One Column Per Line

```
SELECT
 a
,b
,c
FROM <A>
WHERE <TRUE>
```

not

```
SELECT a,b,c
FROM <A>
WHERE <TRUE>
```

Why put the comma first?

Aligning Column Names

Align column names with manual spaces

```
SELECT
 short_column_name         AS col1
,longer_column_name        AS col2,
,longest_column_name       AS col3,
,short_column_name + 2 * 3 AS col4
FROM <A>
WHERE <TRUE>
```

Nesting Subqueries

If nesting subqueries, use consistent indentation

```
SELECT
 a
,b
FROM (
    SELECT
     c
    ,d
    FROM <A>
    WHERE <TRUE>
)
```

Additional Suggestions

Additional suggestions:

- Capitalize operators, such as SELECT, FROM, WHERE, etc
- Use snake_case for naming columns and subqueries
- Avoid using spaces in names
- Adopt aliases for all tables used, even if only using one table
- Less rewriting to do when (usually not if) you add a second table
- Popular approach is to use first letters of words in table names, such as ct for customer_transactions

Managing a `sqlite` Database

CSV to Database

If you have CSV files, you can create a database like this:

Listing 1 `create_db.py`

```
engine = create_engine("sqlite:////Users/hlukas/git/personal_w

bids = pd.read_csv('/Users/hlukas/Google Drive/Raw Data/goodwi
items = pd.read_csv('/Users/hlukas/Google Drive/Raw Data/goodw

items_small = items.sample(500)
bids_small = bids.loc[bids['itemId'].isin(items_small['itemId

bids_small.to_sql(con=engine, name='bids', if_exists='replace
items_small.to_sql(con=engine, name='items', if_exists='repla
```

Inserting Data Into Table

Listing 2 `update_db.py`

```
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine(f'sqlite:///{path}')
Base = declarative_base()
Base.metadata.create_all(engine)

items = pd.read_csv('/Users/hlukas/Google Drive/Raw Data/goodw

items_small = items.sample(500)

items_small.to_sql(con=engine, name='items', if_exists='append
```

Using `sqlalchemy`

Avoiding Queries

We don't really need to write SQL if we don't want to to use the package:

```
from sqlalchemy import MetaData, Table, select
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine(f'sqlite:///{path}')
Base = declarative_base()
Base.metadata.reflect(engine)
bids = Base.metadata.tables['bids']
query = select(bids.c.itemId, bids.c.bidAmount)\
    .where(bids.c.bidAmount==10)\
    .limit(5)

with Session(engine) as s:
    print(pd.DataFrame(s.execute(query)))
```

```
      itemId  bidAmount
0  177106026       10.0
1  177963226       10.0
2  178438915       10.0
```

Group Operations

```python
from sqlalchemy import func, distinct

query = select(
    bids.c.itemId,
    func.count(distinct(bids.c.bidderName)
).label('n_bidders'))\
    .group_by(bids.c.itemId)

with Session(engine) as s:
    print(pd.DataFrame(s.execute(query)))
```

```
        itemId  n_bidders
0    165561698          1
1    170983900          1
2    172998011          2
3    173907435          1
4    174445924          3
..         ...        ...
167  182760698          1
168  182777527          1
```

Distributed Computing

Distributed Computing and SQL

- One benefit of knowing SQL is that it gives us access to database solutions that facilitate parallelized operations
  - For example, IBM Netezza or Spark
- If our data is big, having the database parallelize operations makes our lives much easier

Non-Parallel Computing

- Simple example
  - I have the vector `[1,2,3,4,5]`
  - I want to square each element
- This requires five computations
- Suppose each computation takes x seconds
- If I run this computation on one "computer", it will take roughly $5x$ seconds to compute

Parallel Computing

- Suppose now I have five computers available
- If the "overhead" to coordinate the tasks is t (sending out the instructions and getting back the results), then parallel computing is an improvement if

$$5x \geq x + t \iff t \leq 5$$

Parallel Computing

- Netezza and Spark (and many others) handle a lot of this on their own
- What should we consider outside of the overhead cost when considering running code in parallel?
  - Is the task actually parallelizable?
  - How many cores should I allocate to the task?

Appendix

Luke Wylie